HTL Leonding
next level

HÖHERE TECHNISCHE
BUNDESLEHRANSTALT
LEONDING

3D Portfolio Gallery – Server

DIPLOMARBEIT

verfasst im Rahmen der

Reife- und Diplomprüfung

an der

Höheren Abteilung für IT-Medientechnik

Eingereicht von:
Ema Halilovic

Betreuer:
Patricia Engleitner
Natascha Rammelmüller

Leonding, März 2024

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

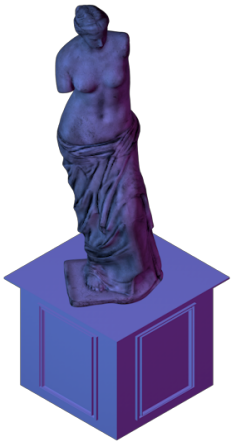
Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Leonding, Dezember 2023 Ema Halilovic

Abstract

3D Portfolio Gallery – Server is the back end of the application *3D Portfolio Gallery*, which was developed by Litzlbauer Lorenz and Maar Fabian. Thanks to it, designers get the possibility to upload their work online, where it gets displayed in a three-dimensional room.

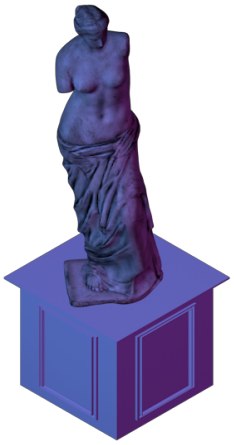


The server is developed by Halilovic Ema and provides an interface to provide the front end with all the needed data. This data is saved in a database, which also enables file upload. The programmed interfaces are secured by a token system.

The technology used to build the server was Quarkus, along with a PostgreSQL database. For token management, JSON Web Tokens were used.

Zusammenfassung

3D Portfolio Gallery – Server ist das Backend der Applikation *3D Portfolio Gallery*, welche von Litzlbauer Lorenz und Maar Fabian programmiert wurde. Sie bietet Designern und Designerinnen eine Möglichkeit, eigene Werke innerhalb eines dreidimensionalen Raumes auszustellen.



Der Server wurde von Halilovic Ema erstellt und bietet der vorher erwähnten Arbeit Schnittstellen, um alle Daten abrufen zu können. Die dafür benötigten Daten sind auf einer Datenbank gesichert. Dank dieser wird ebenso das Hochladen von Dateien ermöglicht. Die ausprogrammierten Schnittstellen werden durch ein Token-System verschlüsselt.

Die, für den Aufbau des Servers verwendete Technologie, ist Quarkus, gemeinsam mit einer PostgreSQL-Datenbank. Für die Token-Verwaltung werden JSON Web Token verwendet.

Inhaltsverzeichnis

1 Einleitung

Die genannte Arbeit, erstellt von Litzlbauer Lorenz und Maar Fabian, ermöglicht die Erstellung von interaktiven und dreidimensionalen Räumen. Diese Räume werden mittels Angular in einem Browserfenster abgebildet.

Als Ausgangsbasis wurde die Diplomarbeit *3D Portfolio Gallery* herangezogen und mit folgenden Funktionalitäten erweitert:

- Implementierung von REST-Schnittstellen
- Hosten der Webseite innerhalb einer Cloud

Die Bereitstellung der benötigten Daten in Form eines Backends, ist die Hauptbeschäftigung dieser Diplomarbeit.

1.1 Zielsetzung

Das Hauptziel dieser Arbeit ist es, eine API zu erstellen, welche die Kommunikation zu serverseitigen Daten ermöglicht. Dafür ist eine Abbildung der Datenstruktur in einer Datenbank erforderlich. Damit diese fehlerfrei erfolgt, wird ein ERD verwendet.

2 Technologien

Bei der Auswahl der Technologien für den serverseitigen Bereich wurden folgenden Kriterien berücksichtigt: Die Auswahl der Technologien für den serverseitigen Bereich berücksichtigt die Erfüllung folgender Kriterien:

- Sie sollen auf dem neuesten Stand der Technik sein
- Ausreichende Dokumentation mit ständiger Weiterentwicklung verfügen
- Eine weite Auswahl an Frameworks und Funktionalitäten unterstützen in den Bereichen:
  - REST
  - Filemanagement
  - Datenbank

Neben der Erfüllung der oben genannten Kriterien wurden bei der Entscheidung zur Verwendung der Technologien auch bereits vorhandene Praxiserfahrungen berücksichtigt. Eine Übersicht sowie nähere Beschreibung der für die vorliegende Arbeit verwendeten Technologien sind in den folgenden Unterkapiteln gegeben. Allerdings wurden die endgültigen Technologien aufgrund schon vorhandener Praxiserfahrung getroffen.

2.1 PostgreSQL

PostgreSQL ist ein Open Source System für relationale Datenbanken. Es wurde 1986 erstmals an der University of Califorinia at Berkeley entwickelt, was eine Entwicklung seit 35 Jahren bedeutet. *PostgreSQL* läuft auf den gängigsten Betriebssystemen im Terminal. Durch die Entwicklung des Systems seit rund 35 Jahren, entstanden ebenso mehrere User-Interfaces, wie beispielsweise PgAdmin. [14]

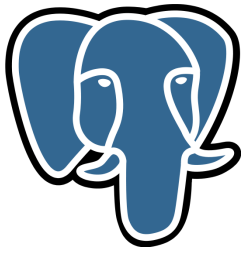


Abbildung 1: PostgreSQL Logo

PgAdmin ist das am weitesten verbreitete Administrationstool für PostgreSQL. [15] Es erleichtert die Ausführung von SQL-Befehlen, sowie den Wechsel verschiedener Tabellen.

Eine gute Dokumentation und schon vorhandene Praxiserfahrung haben diese Datenbank hervorgehoben und zur endgültigen Entscheidung geführt. In dieser Arbeit wurde PgAdmin 4 verwendet, um Daten mittels Port-forwarding auf der Cloud zu überprüfen. Durch die Visualisierung der Tabellen war diese Aufgabe einfacher als mit dem Terminal allein.

2.2 Quarkus

Das Open Source Framework *Quarkus* wird für die Entwicklung von cloud-native Projekten in Java verwendet. Es zeichnet sich unter anderem durch die kurzen Startzeiten, sowie den geringen Arbeitsspeicherverbrauch aus. Ebenso sind Quarkus-Projekte dazu optimiert, in Containerumgebungen, wie beispielsweise *Kubernetes*, ausgeführt zu werden. [1, 2]

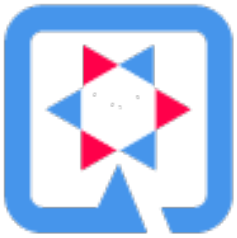


Abbildung 2: Quarkus Logo

Ein neues Projekt wird entweder durch das Quarkus-CLI erstellt oder nach individueller Konfiguration auf code.quarkus.io heruntergeladen. Standardmäßig sind in dem Projektordner Maven-Verzeichnis, sowie Quarkus-Dateien enthalten, welche Projekteinstellungen enthalten. Eine dieser Konfigurationsdateien wäre zum Beispiel die *application.properties*-Datei. [2, 3]

Zusätzlich verfügt Quarkus über eine Vielzahl von Extensions, welche die gängigsten Technologien unterstützen, wie zum Beispiel einen JDBC-Treiber (siehe 2.2.2) oder REST-Easy (siehe 2.2.4). Sie werden durch das CLI oder manuell hinzugefügt. [2, 3]

Mit Quarkus wurde der Backend-Teil der Arbeit erstellt. Die Möglichkeit das Projekt mit zusätzlichen Erweiterungen auszustatten war

2.2.1 Maven

Maven ist ein Open Source Build-Tool, welches von der Apache Software Foundation lizensiert ist. Es wurde in Java geschrieben und wird meist in Java-Projekten eingesetzt. Durch zentrale Nutzung des POM werden projektrelevante Informationen gesichert, wodurch der Kompilierungsprozess vereinfacht wird. [4, 5]

Abbildung 3: *Apache Maven* feather Logo

Maven sorgt dafür, dass unterschiedliche Geräte eine Applikation ausführen können, ohne manuelle Konfiguration. Die einzige Voraussetzung des Zielgerätes ist, dass *Maven* installiert und eingerichtet ist. Ebenso können *Maven-Ordner* gedownloadet werden. Diese ermöglichen die Nutzung der Maven Befehle ohne Installation des Tools am Gerät. [4, 5]

Quarkus-Projekte verwenden *Mavens pom.xml*-Datei, um unter anderem die verwendete Java Version oder die verwendeten Extensions abzuspeichern. Zusätzlich wird ein einheitliches System für Projektkonfigurationen geboten. Dadurch müssen, wie oben erwähnt, Einstellungen bei Gerätewechsel nicht mehr manuell geändert werden. [4]

2.2.2 JDBC Treiber

In Java wird für Datenbankverbindungen die JDBC API verwendet. Genauso benötigen Quarkus-Projekte einen datenbankspezifischen JDBC-Treiber für einen Zugriff auf die Datenbank. In diesem Projekt wurde PostgreSQL als Datenbank gewählt, weshalb die Extension *JDBC Driver - PostgreSQL* fundamental für die Entwicklung war. [6].

Um die Extension zu verwenden und eine Datenbankverbindung aufzubauen, müssen in den *application.properties* die zusätzliche Konfigurationen aus Codeblock 1 eingefügt werden. Wichtig sind Informationen, wie die Art der Datenbank, der Pfad, um diese zu erreichen, und die Login-Daten eines berechtigten Users. Dazu wird der Boilerplate Code aus Codeblock 1 verwendet, wobei die Klammern durch tatsächliche Werte ersetzt werden. [6]

Listing 1: Beispielkonfigurationen

```
1    quarkus.datasource.db-kind=postgresql
2    quarkus.datasource.username=<meinUser>
3    quarkus.datasource.password=<meinPassword>
4    quarkus.datasource.jdbc.url=jdbc:postgresql://<URL>:<Port>/<meinName>
```

2.2.3 Hibernate ORM mit Panache

Da Java eine objekt-orientiert Programmiersprache ist, und PostgreSQL eine relationale Datenbank, können standardmäßig keine Entitäten aus Java in die Datenbank übertragen werden. Diese Aufgabe übernimmt ein ORM, der den Java-Code für PostgreSQL umwandelt. Dadurch können Java-Klassen als Objekte persistiert und gelöscht werden, ohne dass komplexer Code konstruiert werden muss. Hibernate ORM ist die dafür zuständige Extension für Quarkus. [7]

Panache bietet zusätzliche Klassen mit Funktionen, von denen abgeleitet werden kann. Diese erleichtern das Arbeiten mit angelegten Entitäten, besonders bei CRUD-Operationen. Zum Beispiel ist für die Persistenz eines neuen Objekts lediglich ein Aufruf der *.persist()*-Methode notwendig, wodurch automatisch eine Entität in die Datenbank eingefügt wird. [8]

2.2.4 REST-Easy

REST-Easy ist eine Quarkuserweiterung, die es ermöglicht, mit RESTful Web Servies zu arbeiten. Sie implementiert die Jakarta RESTful Web Services der Eclipse Foundation, sowie die MicroProfile REST Client Spezifikation API. [9]

Abbildung 4: REST Easy Logo

Dies bedeutet, dass durch diese Extension im Projekt APIs erstellt werden können. Ein Endpoint wird durch Definition zweier Annotationen *@Path* und *@<beliebige HTTP-Methode>* erzeugt. [10]

Folgende HTTP-Methoden werden in dieser Arbeit verwendet:

- **GET**: GET-Requests liefern Daten [11]
- **POST**: POST-Requests übermitteln Daten wie Entitäten [11]
- **DELETE**: DELETE-Requests löschen Daten [11]

Die Beschreibungen der oben angeführten Methoden sind Normen in der Entwicklung mit HTTP. Eventuell hätte noch *PUT* verwendet werden können. Diese Methode wird ähnlich wie das *POST* verwendet, jedoch ist es ausgelegt für einzelne Änderungen einer Entität. Dadurch hatte sie in dieser Arbeit keinen spezifischen Nutzen. [11]

2.2.5 Swagger-ui

Swagger UI ist ein Tool zum Testen von REST-APIs. Es ist Open Source und in dieser Arbeit hauptsächlich die Visualisierung der Schnittstellen verwendet. Dieser Teil bietet, je nach definierten Parametern, vorgefertigte Requests, welche noch bearbeitet werden können. Ebenso zeigen sie die benötigte HTTP-Methode und ermöglichen es, den Endpoint mit Mausklick zu testen. Dies erspart sehr viel Arbeit während der API-Entwicklung. [12]

Abbildung 5: Swagger Logo

2.2.6 JUnit 5

JUnit steht für Java Unit und ist ein Open Source Framework. Es hat sich als der Standard zum Testen von *Java*-Programmen bewiesen und ist spezialisiert auf die

Überprüfung von Methoden und Klassen. Durch die Popularität besteht eine ausführliche Dokumentation und weites Funktionsspektrum. Ebenso inspirierte dieses Konzept die Frameworks anderer Programmiersprachen im Testing-Bereich.

JUnit ist in vielen der gängigsten IDEs inkludiert. Zum Anlegen einer Testklasse wird gewöhnlich der Name der zu testenden Klasse verwendet mit *.test* als Suffix. Die benötigten Repositories werden simuliert, sodass Methoden aus diesem wie gewohnt verwendet werden können. [13]

2.3 IntelliJ IDEA

IntelliJ IDEA ist eine IDE, welche von JetBrains entwickelt wurde. Diese ist ausgelegt für Java- und Kotlin-Projekte und assistiert bei verschiedensten JVM-Frameworks. Durch eingebaute Features erleichtert diese Entwicklungsumgebung das Programmieren für Nutzende. Plug-Ins ermöglichen es, Datenbankverbindungen und weiteres in der IDE zu konfigurieren, sodass dem Entwickler oder der Entwicklerin eine Übersicht von benötigten Informationen gegeben werden kann. [16]

Abbildung 7: IntelliJ Logo

IntelliJ ist nicht Open Source und benötigt ein kostenpflichtiges Abonnement. Als Schüler und Schülerin kann nach der Angabe von Schulinformationen eine gratis Lizenz für 6 Monate alle JetBrains Produkte beantragt werden. Wäre der Benefit nicht vorhanden, hätten wir die IDE nicht gewählt, sondern Visual Studio Code 2.4 für den ganzen Entwicklungsprozess verwendet. [17]

2.4 Visual Studio Code

Visual Studio Code ist ein Open Source Code Editor für verschiedene Zwecke. Das Programm ist auch unter dem Namen *VS Code* bekannt und wurde von Microsoft lizensiert. Der offen verfügbare Quellcode führt dazu, dass die Community weiterhin Erweiterungen entwickelt. [18]

Abbildung 8: VS Code Logo

Der Editor läuft auf den Betriebssystemen Windows, macOS und Linux. Bei der Installation wird anfangs nur ein Language Support für JavaScript, TypeScript und Node.js mitgeliefert. Durch die vielen Erweiterungen kann sich VS Code individuell angepasst werden. Ebenso kann es als IDE genutzt werden, wenn eine Extension für die gewünschte Sprache gefunden werden kann. [19]

Der Editor wurde für die letzten Schritte der Arbeit genutzt. Dazu zählen die Konfiguration der benötigten Cloud-Dateien, sowie das von Frondend und Backend.

2.5 LeoCloud

Die LeoCloud ist ein Projekt der HTL Leonding mit Aberger Christian als Projektleiter. Sie ermöglicht es mittels Kubernetes, eine beliebige Anwendung auf einer Cloud laufen zu lassen. Um diese Cloud nutzen zu können, ist eine E-Mail-Adresse mit der HTL-Leonding-Schulsignatur notwendig. Da dies ein schulinternes Projekt ist, war eine Kommunikation mit den Entwicklern gegeben, das heißt, dass Fragen sofort beantwortet werden konnten. [20]

Die Cloud wurde für die Anforderungen genutzt, dass Projekt für alle Geräte zugänglich zu machen. Durch ein Deployment auf die LeoCloud war die Anwendung über den Browser erreichbar und somit ein endgültiges Produkt.

2.5.1 Kubernetes

Kubernetes ist eine Anwendung, die für das Containern von Applikationen entwickelt wurde. Sie ist Open Source und weit verbreitet. Ebenso basiert sie auf den Produktion-workloads von Google. Alternativ ist Kubernetes unter dem Namen *k8s* bekannt. [21]

Abbildung 9: Kubernetes Logo

Wie vorhin erwähnt, nutzt die LeoCloud Kubernetes, um Anwendungen auf die Cloud zu laden. Besonders wird dabei das CLI verwendet.

Da das Arbeiten am Terminal unübersichtlich werden kann, wurde ebenso die Extension *Kubernetes von Microsoft* in Visual Studio Code genutzt. [22]

2.6 GitHub

GitHub ist eines der am weitest verbreiteten Entwicklertools auf der Welt. Es ist Open Source verfügbar und ermöglicht mit einer Webseite // TODO Zusätzlich existieren bereits diverse Apps und Extensions, die die Entwicklung mit GitHub auf anderen Plattformen verbreiten. [23]

Abbildung 10: GitHub Logo

Durch GitHub Repositories wird das Konzept der SSOT ermöglicht. Bei der Erstellung von Repositories werden

mehrere Funktionen geboten, wie Pull Requests. Wenn
ein Teammitglied eine Änderung des Codes veröffentlicht, überschreiben diese nicht den aktuellen Stand, sondern erzeugen Pull Requests. Nach einer Überprüfung dank der integrierter Codevorschau können die Requests geschlossen und die neuen Zeilen mit schon vorhandenem Code zusammengeführt werden. Alle Änderungen bleiben in Changelogs gespeichert, sodass immer auf alte Versionen zugegriffen werden kann. [23]

2.7 Notion

Notion ist eine Art digitales Notizbuch, womit so genannte *Workspaces* erstellt werden können. Es ist im Browser, als App in Windows-, MacOS- oder auf iOS- und Android-Geräten verfügbar. Workspaces aufgebaut durch mehrere Seiten, welche durch Befehle erstellt werden. Diese werden auch direkt miteinander verlinkt. Seiten lassen sich mit beliebigem Inhalt füllen und durch einfache */-Befehle* ist es möglich z. B. ein Kanban Board erstellen. [24]

Abbildung 11: Notion Logo

Notion ist gut für Neueinsteiger, da alle Funktionalitäten gut beschrieben sind und es benutzerfreundlich gestaltet ist. Workspaces lassen sich mit anderen teilen, wodurch jeder User auf eine SSOT Version des Workspaces zugreifen kann. Für die Bearbeitung in echt-zeit wird Internet benötigt, falls dieses jedoch ausfällt, werden die Änderungen gespeichert und synchronisiert, sobald wieder eine Netzwerkverbindung aufgebaut wurde. [24]

Das Feature von geteilten Workspaces wird in dieser Arbeit als Erleichterung der Kommunikation zwischen Teammitgliedern genutzt, um gemeinsame Notizen und Termine festzuhalten. Durch die Verwendung dieses Tools wurde sichergestellt, dass jedes Mitglied eine SSOT-Version der geteilten Informationen hatte. Für die selbstständige Arbeit wurde das Tool ebenso genutzt, um wichtige Webseiten zu speichern, sowie Zwischennotizen zu erstellen.

2.8 PlantUML

PlantUML ist ein Tool, welches die Erstellung von UML-Diagrammen vereinfacht. Das Ziel von UML ist es, Tools zu liefern, um Systeme zu visualisieren. [25] Dafür gibt es verschiede Arten von UML-Diagrammen, wie beispielsweise Objektdiagramme oder Sequenzdiagramme. [26]

Abbildung 12: PlantUML Logo

Für diese Arbeit wird die gratis Extension *PlantUML von jebbs* verwendet. Sie bietet eine Preview des Diagramms, sowie Exportmöglichkeiten. [27] Die Visualisierung unserer Datenstruktur geschieht durch die Erstellung eines ERDs mit den PlantUML Klassendiagrammen. Die Beziehungen zwischen den Entitäten werden hier mittels der Krähenfußnotation dargestellt.

2.9 JSON Web Token

JSON Web Token, oder JWT, ist ein, durch RFC 7519 gekennzeichneter, Standard für die sichere Datenübertragung. Die Token übermitteln Daten als digital signierte JSON-Objekte. Die Signatur erfolgt durch eine Verschlüsselung mittels Secret oder Schlüsselpaar. Secrets werden mit einem HMAC-Algorithmus erstellt, wobei Schlüsselpaare RSA oder ECDSA verwenden [28]

Abbildung 13: JWT Logo

In Tokens kann eine freie Anzahl von wählbare Daten, wie selbst definierte Adjektive, gespeichert werden. Typischerweise werden Tokenart und Verschlüsselungsalgorithmus im Header des Tokens gespeichert und die individuell definierten in der *Payload*. Der Token wird bei einem erfolgreichem Login Request erhalten und danach bei HTTP-Requests im Authorization Header mitgeschickt. Damit der Token nicht unendlich lang genutzt werden kann, ist es wichtig, eine TTL festzulegen. Bei Interesse an den gespeicherten Informationen kann jeder JWT auf https://jwt.io enkodiert werden. [28]

2.10 Homebrew

Homebrew ist ein Paketmanager für Linux und macOS. Er wurde in Ruby von Max Howell programmiert und ver-

einfacht Installation und Aktualisierung von Tools und Dependencies. Dazu müssen sie im Homebrew-Repository enthalten sein. Alle Installationen werden im selben Ordner gespeichert, wodurch bei Bedarf des Pfades dieser sofort gefunden wird. Um Homebrew zu verwenden, wird im Terminal Stichwort *brew* genutzt.

Die meisten der oben angeführten Technologien sind im Homebrew-Repository enthalten. Durch die Entwicklung auf einem macOS-System, war dieser Paketmanager ein sehr hilfreiches Tool. [29]

3 Umsetzung

In diesem Kapitel wird die Umsetzung der Anforderungen näher beschrieben. Neben Erklärungen finden sich ebenso Codeausschnitte und Befehle, um den Lösungsweg zu veranschaulichen. Zusätzlich wurden Grafiken eingebettet, um den technischen Aspekt visueller darzustellen.

3.1 Design und Planung

Der Designprozess dieser Arbeit handelt überwiegend von dem Aufbau des Datenmodells. Um dieses zu Visualisieren wurde ein ERD gewählt.

3.1.1 Allgemeine Anforderungen

Die ersten teaminternen Gespräche handelten von den ersten Anforderungen der Planung. Es wurde sich auf eine Erstellung eines Workspaces in Notion geeinigt, sowie ein gemeinsames GitHub-Repository. Dadurch wurde für Termine von Meetings und Deadlines, sowie den Designentscheidungen in beiden Bereichen eine SSOT gewährleistet. Ebenso kamen die ersten Anforderungen der Datenstruktur zustande, welche für das zukünftige ERD relevant waren. Die tatsächliche Entwicklung des Applikationsservers fand in einem separaten Repository statt.

Anforderungen an die Applikationen wurden als User Stories mit Akzeptanzkriterien definiert. Zusätzlich fasste ein ERD das benötigte Datenmodell zusammen.

3.1.2 Erstellung eines Workspaces in Notion

Um auf Notion zuzugreifen und Workspaces zu verwalten, wird ein Account benötigt. Nach erfolgreicher Anmeldung erfolgt die Erstellung eines Workspaces mithilfe des Tutorials auf der Webseite. Einer der ersten Schritte ist, auszuwählen, für welchen Zweck der Workspace genutzt wird. Nachdem die Option für Teams ausgewählt wurde, können die E-Mail-Adressen der gewünschten Mitglieder eingegeben werden. Diese bekommen einen Link zugeschickt, den sie öffnen können und somit auf den Workspace

gelangen. Um ein Kollaborator zu werden, muss ebenso bei den eingeladenen Personen ein Notion-Account bestehen.

Der Workspace der Arbeit wird in Abbildung 15 gezeigt. Das rosa Rechteck zeigt die Teilnehmer an, wobei ihre genauen Informationen erst mit einem Mousehover ersichtlich werden.

Da *3D Portfolio Gallery – Server* anfangs im Team entwickelt wurde, war es nützlich einen Workspace zu erstellen um Notizen nach Meetings zu teilen und gemeinsame Termine festzuhalten. Ebenso wurden wichtige Links abrufbereit gespeichert und erste Dokumentationen der schriftlichen Arbeit erstellt. Nach Aufteilung des Teams, wurde der Workspace weiterhin zu selbstständiger Dokumentation genutzt.

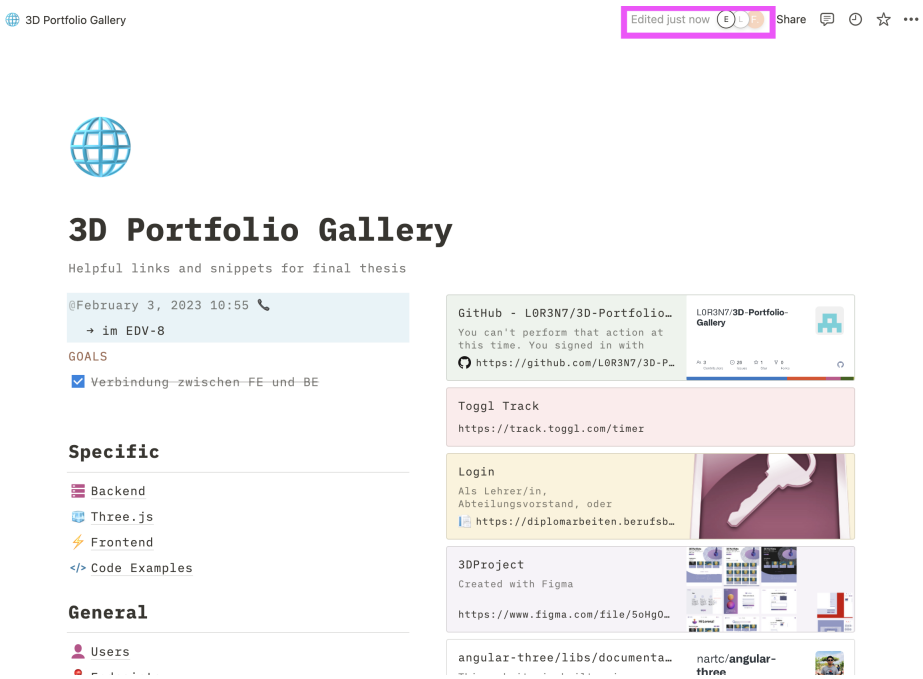


Abbildung 15: Notion Workspace der Diplomarbeit

3.1.3 Erstellung des ERDs

Um zu gewährleisten, dass die Datenanforderungen getroffen werden, ist es notwendig, ein entsprechendes ERD zu erstellen. Innerhalb eines ERDs werden alle Entitäten, Attribute, sowie deren Beziehungen zueinander dargestellt. Es dient als gemeinsamer Nenner für die Projektentwicklung, da jedes Teammitglied eine eigene Vorstellung des genauen Datenaufbaus haben könnte. Ein weiterer Vorteil eines ERDs ist das Potenzial, Fehlerquellen zu identifizieren und dadurch zukünftige Probleme zu umgehen.

Nachdem die benötigten Entitäten, sowie deren Attribute festgelegt wurden, müssen nun die Beziehungen zueinander festgelegt werden. Dabei wird gewählt zwischen One-to-One, One-to-Many und Many-to-Many Beziehungen, wobei die Many-to-Many Beziehung innerhalb einer Assoziationstabelle dargestellt wird.

Mittels PlantUML wurde das ERD erstellt, wobei es nur die Visualisierung bietet. Es hilft nicht zusätzlich bei der Erstellung durch beispielsweise Codecompletion. Für PlantUML wird Datei mit unterstützter Endung benötigt, typischerweise mit *.plantuml*. Im File müssen die Annotationen in *@startuml* und *@enduml* enthalten sein. Entitäten werden mittels *entity* gekennzeichnet. Beziehungen werden nebeneinander geschrieben, mit einem Strich als Trennung. In der Krähenfußnotation besitzen diese Striche über verschiedenen Endungen, je nach Beziehungstyp.

Listing 2: Ausschnitt der ERD-Datei

```
@startuml DA-Klassendiagramm
entity Exhibition_Category {
    exhibition_id : number
    category_id : number
}

entity Category{
    id : number <<generated>>
    --
    title: string
    color: varchar2(6)
}
-- ...
Exhibition_Category ||--o{ Category
@enduml
```

Innerhalb dieser Arbeit ist die Entität *Exhibitions* das zentrale Objekt, was in der Abbildung 17 ersichtlich ist. Die Beziehungen zwischen *Positionen, Exhibits, Rooms und Exhibitions* stellten sich besonders als Herausforderung dar, aufgrund der vielen Abhängigkeiten zueinander. Diese wurden besonders erst in der Implementierung ersichtlich, wodurch das ERD im Laufe der Entwicklung entsprechend angepasst wurde. Weitere Gründe für die Abänderung, waren Planänderungen bei der Kommunikation zwischen zwei Entitäten, sowie der Aufbau der Hauptentitäten *Exhibition und Exhibit*. Die Unterschiede zwischen der ersten und der finalen Version sind deutlich (siehe Abb. 16 und 17). Die Tabelle *available_Exhibits_in_Exhibitions* wurde entfernt, während eine neue Mechanik mittels *Themes* hinzugefügt wurde. Statt erster Tabelle wurde eine direkte Verbindung zwischen *Exhibit* und *Position* aufgestellt. Die Attribute des *Exhibits* wurden ausgearbeitet, sowie eine *Eins-zu-Viele*-Beziehung zur Entität *Exhibition* aufgebaut. Ebenso wurde der *User* nicht mehr einzelnen *Exhiibits* zugeteilt, sondern nur keiner oder mehrerer *Exhibitions*. Dieser Schritt war in der Hinsicht wichtig, um Daten bei Löschvorgängen konsistent zu halten.

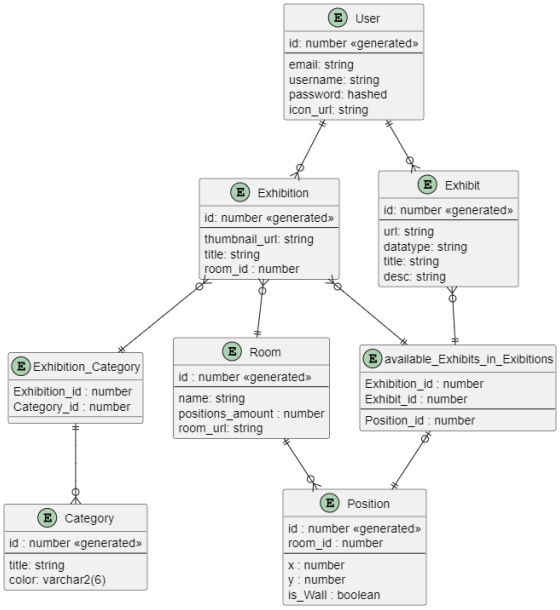


Abbildung 16: Erste Version des ERDs

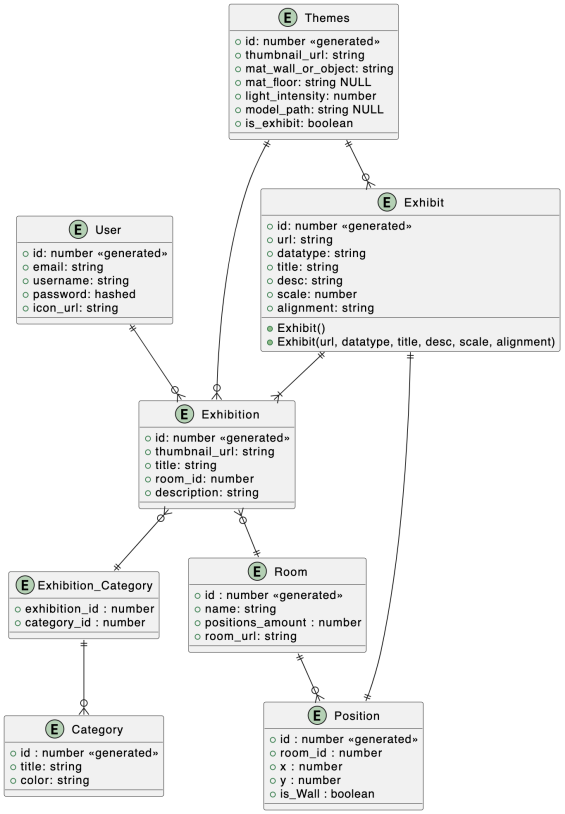


Abbildung 17: Finale Version des ERDs

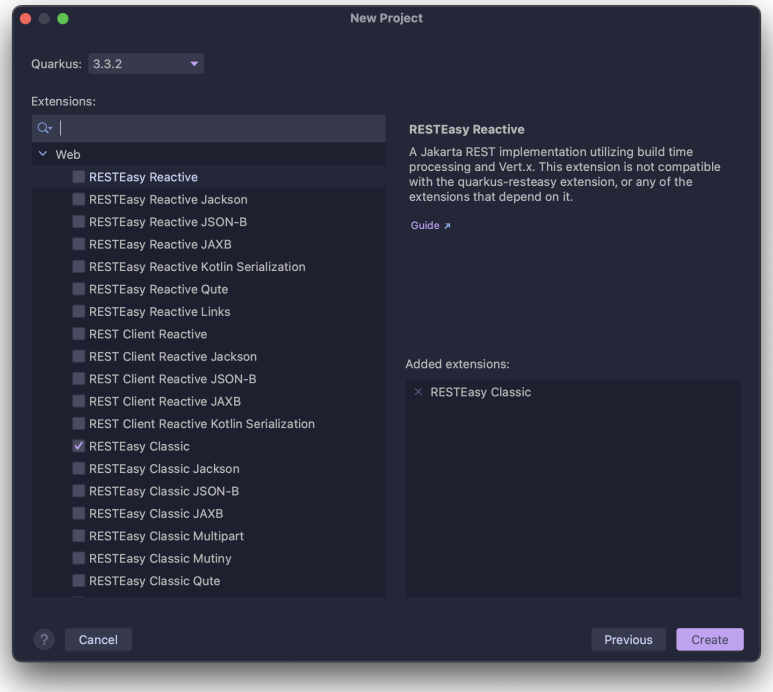


Abbildung 18: Auswählen der gewünschten Quarkus Extensions im Quarkus Plugin

3.2 Aufsetzen der Datenbank und des Servers

Um das Quarkus-Projekt aufzusetzen, gibt es mehrere Möglichkeiten. Online gibt es die Möglichkeit unter code.quarkus.io alle gewünschten Extensions mit Mausklick auszuwählen und anschließend das generierte Projekt herunterzuladen. Am Terminal besteht die Optionen mittels Maven oder Quarkus-CLI-Befehl ein Projekt zu erstellen, wobei gewünschte Erweiterungen händisch eingetippt werden müssen.

Für diese Arbeit wurde jedoch das IntelliJ Plug-In *Quarkus* von *JetBrains s.r.o.* verwendet. Dieses bildet dieselben Möglichkeiten, wie die Webseite, in einem griffbereitem IDE-Fenster ab (siehe Abb. 18). Dadurch muss kein weiteres Programm, wie ein Browser, gestartet werden nur für die Projekterstellung. Zusätzlich werden Tools zur Verfügung gestellt, wie zum Beispiel Code-Assistenz, oder Laufkonfigurationen. [30] Diese Gründe führten zu seiner persönlichen Präferenz dieser Methode, anstelle der anfangs erwähnten Optionen.

Durch die Verwendung von Maven werden alle Extensions im *pom.xml*-File des Projekts gespeichert. Falls Erweiterungen im Nachhinein noch hinzugefügt werden sollen, kann dies durch Terminal Befehle automatisch geschehen, oder durch manuelles Einfügen in die Datei. Bei letzterem muss in den meisten Fällen ebenso die Maven manuell neu geladen werden, da Änderungen sonst nicht übernommen sein könnten.

Listing 3: Quarkus-CLI-Befehl für RESTEasy Classic

```
quarkus ext add io.quarkus:quarkus-resteasy
```

Listing 4: Extension RESTEasy Classic in pom.xml

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy</artifactId>
</dependency>
```

Nach der Erstellung des Projektes, sollte es sofort ausgeführt werden, um sicherzustellen, dass es keine fundamentalen Probleme am Projekt gibt. Dazu wird das *developer* Profil von Quarkus genutzt, was mehr Funktionen ermöglicht, wie zum Beispiel das Loggen der Konsole. IntelliJ konfiguriert die den Befehl automatisch, sodass dafür nur ein Buttonklick zuständig ist. Alternativ kann auch das Terminal verwendet werden.

Listing 5: Quarkus-CLI-Befehl zum Starten

```
quarkus dev
```

Da mit der Anwendung Tabellen erstellt werden sollen, wird eine Datenbankverbindung benötigt. Dazu wird PostgreSQL am Gerät installiert und gestartet. Als Erleichterung dieses Schritts wurde *Homebrew* genutzt, wodurch durch die unten angeführten Befehle die Installation und Ausführung automatisch geschah.

Listing 6: Homebrew Installation und Ausführung PostgreSQL

```
brew install postgres
brew services start postgres
```

Als nächsten Schritt wird eine Verbindung von der Quarkus Applikation zu PostgreSQL aufgebaut. Name, Rolle bzw. User und Port der Datenbank können frei konfiguriert werden, allerdings wurden hier die standardmäßig definierten Werte gelassen. Für den Verbindungsaufbau während der Entwicklung werden die Einstellungen in Abbildung 7 genutzt. Falls die Anwendung auf eine Cloud hochgeladen wird, müssen andere Parameter gesetzt werden.

Listing 7: Datenbankkonfiguration in *application.properties*

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username = postgres
quarkus.datasource.password = postgres
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:5432/postgres

quarkus.hibernate-orm.database.generation=drop-and-create
```

Nachdem die Einstellungen eingefügt wurden, muss die Anwendung neu gestartet werden. Falls der Verbindungsaufbau versagt, wirft Quarkus während der Kompilierung Fehlermeldungen. Andernfalls können nun die Tabellen angelegt werden.

3.2.1 Umsetzung des ERDs in Quarkus

Für den nächsten Schritt wird die Extension *Hibernate ORM mit Panache* verwendet. Mittels dieser werden Tabellen durch Definition der Annotation *@Entity* erstellt. Die Annotation sieht vor, dass in jeder Entity-Klasse eine Id steht. Durch die Ableitung der *PanacheEntity*-Klasse (siehe Abb. 8, Zeile2), die schon eine Id in sich trägt, muss sie nicht mehr explizit definiert werden. Diese Id wurde ebenso als Primärschlüssel genutzt, da sie durch Generierung immer einzigartig ist.

In der Klasse werden alle Attribute als Variablen, gemäß des ERDs definiert. Die Beziehungen werden durch die Annotationen *@OneToMany, @ManyToMany und @OnetoOne* über der dazugehörigen Variable dargestellt. Auf der gegenseitigen Entität muss ebenso eine Variable erstellt werden, wobei da die Annotation etwas anders aussieht. Diese muss die *mappedBy*-Einstellung konfigurieren, da ansonsten die Beziehung mehrmals generiert wird (siehe Abb. 8, Zeile 4). Bei Many-to-Many wird zusätzlich die Annotation @JoinTable hinzugefügt, um die Assoziationstabelle zu konfigurieren (siehe Abb. 8, Zeilen 5 bis 12). Zu den Relationen können zusätzlich noch Cascade Types festgelegt werden. Dessen Funktionen werden in Unterkapitel Cascade Types genauer erklärt.

Listing 8: Teil der Exhibition Entität

```
1  @Entity
2  public class Exhibition extends PanacheEntity {
3      // ...
4      @OneToMany(mappedBy = "exhibition", cascade = CascadeType.REMOVE)
5      public List<Exhibit> exhibits;
6
7      @ManyToMany
8      @JoinTable(
9              name = "exhibitions_categories",
10             joinColumns = @JoinColumn(name = "exhibition_id"),
11             inverseJoinColumns = @JoinColumn(name = "category_id")
12     )
13     public Set<Category> categories;
14 }
```

Nachdem die Klassen komplett sind, muss die Anwendung neu gestartet werden. In den *application.properties* wurde durch *drop-and-create* konfiguriert, dass alle Tabellen bei Neustart gelöscht und neu aufgesetzt werden. Diese Einstellung sorgt dafür, dass jede vorgenommene Änderungen tatsächlich übernommen werden.

Ebenso wird bei jedem Start eine *import.sql*-Datei aufgerufen, die vordefinierten Daten bei jedem Start automatisch in die Datenbank lädt. Für die Entitäten *Theme*, *Positionen* und *Rooms* hat sich das Team dazu entschieden, diese vorzugeben. Andere Entitäten werden als Testdaten in das File eingefügt und besitzen negative Ids. Die negativen Werte stellen sicher, dass keine Objekte in die Datenbank geladen werden, mit demselben Primärschlüssel.

Listing 9: Ausschnitt import.sql

```
1 insert into category(id, category_title, color) values (1, 'Umwelt', '#C1BAFF'),
      (2, 'Tiere', '#ADDOFF');
2 insert into exhibit(id, title, description, scale, alignment, url, exhibition_id,
      position_id, data_type) values (-20, 'Art Blob', 'Sculpture made by Lorenz
      Litzlanze', 40, 'c', 'example-exhibits/abstraktArtBlob.gltf', -3, 2, 'gltf');
```

User

Die Entität eines Nutzenden, wird User genannt, wobei für den Tabellennamen in der Datenbank jedoch eine alternative Bezeichnung definiert ist. Grund dafür ist, dass der Name *User* in PostgreSQL reserviert ist. Deshalb ist in Codeausschnitt 10 die Zeile 2 erforderlich.

Jeder neuer Benutzer muss einen Nutzernamen und ein Passwort besitzen. Die Id wird automatisch generiert. Mittels der Attribute der *@Column* Annotation, lassen sich obligatorische Felder definieren. Ebenso können Feldlänge, Spaltenname und vieles mehr auf diese Art konfiguriert werden.

Listing 10: Teil der Entity-Klasse des Users

```
1  @Entity
2  @Table(name="Users")
3  public class User extends PanacheEntity {
4      @Column(nullable = false, length = 50)
5      public String user_name;
6      @Column(nullable = false)
7      public String password;
8      public String email;
9      //...
10 }
```

Cascade Types

Cascade Types werden für Datenkonsistenz benötigt, da manche Objekte von anderen abhängen. Die verschiedenen Types entscheiden, was bei der Änderung eines Teils der Abhängigkeit passiert. Es gibt verschiedene Arten von Cascade Typen, dieser Arbeit hauptsächlich die unten aufgelisteten verwendet wurden. [31]

- **CascadeType.ALL**: Alle Operationen werden auf die andere Abhängigkeit übertragen [31]
- **CascadeType.REMOVE**: Wenn der Elternteil der Beziehung gelöscht wird, passiert dasselbe beim Kindelement [31]

Weitere JPA Cascade Types wären die PERSIST, MERGE, REFRESH und DETACH, wobei Hibernate-spezifische REPLICATE, SAVE_UPDATE und LOCK sind. Eventuell

hätten die Arbeit noch den Typ *PERSIST* einbauen können, da dieser alle Kinderelemente eines Elternobjekts bei dessen Persistenz mitsichert, jedoch war dies nicht notwendig bei unserer Lösung [31]

3.3 Allgemeines

Damit das Frontend auf Daten zugreifen kann, werden Schnittstellen benötigt, die typischerweise in passenden Resource-Klassen erstellt werden. Solch eine Klasse hat die *@Path*-Annotation oberhalb der Klassendefinition. Die Kennzeichnung legt fest, unter welchem Pfad die Endpoints innerhalb dieser Klasse erreichbar sind.

Ebenso kann mittels *@Produces und @Consumes* global festgelegt werden, welche Art von Request sich die Methoden aus dieser Resource erwarten oder absenden. Methoden können diese Definition überschreiben falls nötig. Innerhalb dieser Arbeit wurde am häufigsten bei dieser Kennzeichnung *MediaType.APPLICATION_JSON* verwendet, da JSON-Objekte eine unkomplizierte Entwicklung ermöglichen.

Falls eine Methode sich mittels *@Consumes* einen Wert erwartet, muss dieser als Parameter dieser definiert werden. Dieser ermöglicht den Zugriff auf die übergebenen Werte.

Listing 11: @Consumes Ausschnitt von UserResource

```
1 @Consumes(MediaType.APPLICATION_JSON)
2 @Path("/login")
3 public Response login(UserLoginDTO loginDTO){
4     var password = this.hashPassword(loginDTO.getPassword());
5 }
```

Jede Methode, die eine Schnittstelle darstellt, muss über eine Annotation, mit der zu verwendenten HTTP-Methode verfügen. Optional eine weitere *@Path*-Annotation erstellt werden. Zusätzlich müssen die HTTP-Methoden *POST* und *DELETE* als *@Transactional* gekennzeichnet werden. Das bewirkt, dass die Methode als Transaktion ausgeführt. Eine Transaktion ist ein Block von Anweisungen, der entweder zur Gänze, oder bei einem Fehlschlag gar nicht ausgeführt wird. Bei zweiterem Fall wird die Datenbank auf den letzten Stand vor Aufruf der Methode zurückgesetzt. [32]

Listing 12: Anfang einer Resource Datei

```
1 @Path("/api/category")
2 @Produces(MediaType.APPLICATION_JSON)
3 public class CategoryResource {
4     // ...
5     @GET
6     @PermitAll
7     @Path("/all")
```

```
8       public Response getAllCategories(){
9           // ...
10      }
11  }
```

Bei Schnittstellen können ebenso Parameter definiert werden, die Werte einlesen. Sie werden mittels *@PathParam* und einem Namen erstellt. Zusätzlich muss der Parametername bei der Annotation *@Path* in geschwungene Klammern geschrieben werden.

Listing 13: Abfrage eines Users

```
1       @GET
2       @Produces(MediaType.APPLICATION_JSON)
3       @Path("/{user_id}")
4       public Response getUser(@PathParam("user_id") long id) {
5           // ...
6       }
```

Anders als beim Anlegen der Tabellen, muss bei der Erstellung und Ausarbeitung der API das die Anwendung nicht neugestartet werden. Zur Testung dieser, wurde die Extension *Swagger UI* verwendet. Diese visualisiert die Endpoints und automatisiert die Erstellung von Abfragen, was Zeit spart. Das Verfahren des Frameworks wird jedoch in Abschnitt 3.7.1 näher bearbeitet.

3.3.1 Repositories

Durch eine Java-Entwicklung beeinflusst durch die Schichtenarchitektur, geschieht in Resource-Klassen, kein Datenbankzugriff. Dafür sind Repositories zuständig, die anschließend injiziert werden. [33]

Eine Injektion erstellt eine Abhängigkeit auf einen Translator. Falls die Klasse des Translators nicht existiert, wird ein Fehler geworfen. Damit die Injektion verwaltet und genutzt werden kann, muss sie mittels *@ApplicationScoped* gekennzeichnet werden. [34]

Durch die Verwendung der Panache Extension, implementieren alle Repository-Klassen das *PanacheRepository*. Dieses fügt Methoden für CRUD-Operationen hinzu, wobei die zugehörige Entität zusätzlich in spitzen Klammern angegeben werden muss. Das Repository kann ebenso mit benutzerdefinierten Methoden erweitert werden.

Listing 14: Ausschnitt aus dem Exhibition Repository

```
1   @ApplicationScoped
2   public class ExhibitionRepo implements PanacheRepository<Exhibition> {
3       //...
4       public List<ExhibitionWithUserRecord> listAllExhibitionsWithUserField() {
5           String sql = "select new
6               org.threeDPortfolioGallery.records.ExhibitionWithUserRecord(e,
                    u.user_name, u.icon_url) from Exhibition e join e.user u left join
                    e.categories c";
```

```
7           TypedQuery<ExhibitionWithUserRecord> q = getEntityManager()
8                   .createQuery(sql, ExhibitionWithUserRecord.class);
9           var ret = q.getResultList();
10          return ret;
11      }
12  }
```

Die Repositories ermöglichen ebenso das Definieren von eigenen JPQL- und SQL-Befehlen. JPQL ermöglicht, neben der Nutzung der gewohnten SQL-Keywords, das Definieren von Parametern in Queries, sowie die Verwendung von eigenen Records und DTOs.

3.4 Einrichtung der JWT

Für die Sicherheit der API wird ein Tokensystem verwendet. Dazu wird ein Schlüsselpaar benötigt und die Erweiterungen *smallrye-jwt und smallrye-jwt-build*. [35]

Listing 15: Erstellung eines Schlüsselpaars

```
1  ssh-keygen -t rsa
```

Dieses wird anschließend im Quarkus-Verzeichnis unter dem neu erstelltem Ordner *resources/jwt/* gesichert. Nach Erstellung muss in den *application.properties* noch definiert werden, wo die Schlüssel liegen, die genutzt werden. [35]

Listing 16: JWT Konfigurationen in application.properties

```
1  mp.jwt.verify.issuer=user-jwt
2  mp.jwt.verify.publickey.location=jwt/publicKey.pem
3  smallrye.jwt.sign.key.location=jwt/privateKey.pem
```

Um den Token tatsächlich zu erstellen wird ein Token-Service benötigt. Die Methode *generateJwt()* wird beim Login eines Users aufgerufen um einen Token zu generieren. Das *claim()* sichert gewünschte Attribute, wobei diese optional sind. In diesem Fall wird die Nutzerid gesichert. *Groups()* fügt den Token zu der Rolle *user* hinzu. Durch ein Rollensystem wird der Zugriff auf die Endpoints eingeschränkt, da jeder Rolle die Berechtigungen spezifisch durch *@RolesAllowed* in Quarkus erteilt werden müssen. Zuletzt wird explizit mittels *expiresAt()* ein Verfallsdatum für jeden Token festgelegt. [35]

Listing 17: Methode zum signen von Tokens in JWT-Service

```
1  public String generateJwt(Long id){
2      return Jwt.issuer("user-jwt")
3              .claim("userid", id)
4              .subject("user-jwt")
5              .groups(Set.of("user"))
6              .expiresAt(System.currentTimeMillis() + 3600000).sign();
7  }
```

Die generierten Token mit der obigen Methode können nur auf Endpoints der API zugreifen mit der Definition *@RolesAllowed("user")*. Ohne JWT kann auf alle Requests zugegriffen werden gekennzeichnet mit *@PermitAll*. Auch wenn die Tokens mit eigenem Schlüsselpaar verschlüsselt sind, können sie auf der offiziellen www.jwt.io Seite entschlüsselt werden. [28]

3.5 Verwaltung der User

Für die Verwaltung der Nutzenden wurden folgende Methoden implementiert:

- **GET** getUser(Long id)
- **POST** login(UserLoginDTO loginDTO)
- **POST** postUser(User new_user, @Context UriInfo uriInfo)
- **DELETE** deleteUser(Long id)

Die Methoden *getUser* und *deleteUser* sind relativ simpel. Sie erwarten sich den Primärschlüssel der Entität *User*, suchen diese, und geben diesen entweder zurück, oder löschen die damit verbundene Entität. Die anderen zwei sind etwas komplexer.

PostUser ist die Methode, verwendet zur Erstellung eines neuen Nutzers. Sie erwartet sich ein JSON-Objekt in Form des Beispiels 18 im Body eines POST-Requests. Nach erfolgreicher Anlegung wird *Statuscode 201 Created* zusammen mit einem URI-Pfad zurückgegeben.

Listing 18: PostUser

```
{
  "user_name": "string",
  "email": "string",
  "icon_url": "string",
  "password": "string"
}
```

Die Methode *login* ist ebenso eine POST-Methode. Sie erstellt einen neuen Token für eingeloggte Nutzer. Dazu wird die *generateJwt()*-Methode des JWT-Services aufgerufen mit der Id des mitgegebenen Users als Parameter. Das genaue Verfahren beim Aufruf wird in Abschnitt 3.4 genauer erklärt. Bei erfolgreichem Login wird Statuscode 204 mit dem JWT zurückgegeben. Falls der Nutzer oder die Nutzerin nicht gefunden wurde, kommt Statuscode 404 zurück.

Zusätzlich muss erwähnt werden, dass Passwörter nicht als Klartext in der Datenbank gespeichert werden. Sie bekommen einen Salt angehängt und werden mithilfe von Base64 verschlüsselt.

Listing 19: Hashing des Passwortes

```
1  private String hashPassword(String string){
2      string = string + "yoyoyo";
3      Base64.Encoder encoder = Base64.getEncoder();
4      return encoder.encodeToString(string.getBytes());
5  }
```

3.6 Verwaltung der Exhibitions

Folgende Methoden wurden für die Verwaltung der Exhibition implementiert.

- **GET** getExhibitionById(Long id)
- **GET** getExhibitionsByUser(Long id)
- **GET** getAllExhibitions()
- **GET** getExhibitionsBySearchTerm(String searchTerm)
- **GET** getLastFiveExhibitions()
- **GET** getExhibitionByCategory(Long id)
- **GET** getExhibitionByCategories(String categoryIds)
- **POST** postNewExhibition(AddExhibitionDTO newExhibition)
- **DELETE** deleteExhibitionById(Long id)

Die Implementierung der Methoden *getExhibitionById*, *getAllExhibitions*, *getExhibitionsByUser* und *deleteExhibitionById* war durch die Verwendung der vorgefertigten Panache-Methoden unkompliziert. Für die Restlichen mussten eigene Queries erstellt werden.

Die Methode *getExhibitionsBySearchTerm* bekommt einen String mitgegeben und soll alle Exhibitions zurückgeben, die diesen Begriff im Titel oder im Usernamen enthalten. Das gesuchte Wort muss in einen Parameter umgewandelt werden, welches von der Query verstanden werden kann. Zusätzlich werden dem String im Parameter Prozentzeichen angehängt, da sonst nur exakte Matches gefunden werden. Diese ermöglichen es, Exhibitions zu finden, die den String nur enthalten.

Listing 20: Exhibition Query für getExhibitionsBySearchTerm

```
1  String sql = "select DISTINCT new
       org.threeDPortfolioGallery.workloads.dto.ExhibitionWithUserDTO(e, u.user_name,
       u.icon_url) from Exhibition e " +
2  "join e.user u left join e.categories c " +
3  "where lower(e.user.user_name) like :term or lower(e.title) like :term order by
       e.id desc";
4  TypedQuery<ExhibitionWithUserDTO> q = getEntityManager()
5      .createQuery(sql
6              , ExhibitionWithUserDTO.class);
7  q.setParameter("term", "%" + term.toLowerCase() + "%");
8  return q.getResultList();
```

Die Methode *getLastFiveExhibitions* wählt alle Exhibitions aus, sortiert diese absteigend nach ihrem Primärschlüssel und grenzt den Output auf fünf Objekte ein. Für das Zurückwerfen der Exhibitions nach einer bestimmten Kategorie für *getExhibitionByCategory*, wurden ebenso Parameter in der Query verwendet.

Eine neue Exhibition wird mit der einzigen POST-Methode der Liste erstellt, *postNewExhibition*. Dafür wird ein sehr großes Objekt erwartet (siehe Abb. 22). Bevor die Entität in die Datenbank eingefügt werden kann, müssen alle Id-Felder auf ihre Existenz überprüft werden. Genauso wird kontrolliert, keine Exhibition ohne Exhibits erstellt wird. Erst danach kann das Objekt mittels der Panache-Methode persistiert werden.

Listing 21: Beispiel für neue Exhibition

```
{
    "thumbnail_url": "string",
    "title": "string",
    "description": "string",
    "room_id": 1,
    "user_id": 1,
    "category_ids": [
        1, 2
    ],
    "exhibits": [
        {
            "data_type": "string",
            "description": "string",
            "title": "string",
            "url": "string",
            "scale": 1,
            "alignment": "string",
            "theme_id": 1,
            "position_id": 2
        },{
            "data_type": "string",
            "description": "string",
            "title": "string",
            "url": "string",
            "scale": 1,
            "alignment": "string",
            "theme_id": 1,
            "position_id": 2
        }
    ]
}
```

Die Methode *getExhibitionByCategories* liest eine Liste von Kategorie-Ids ein und gibt alle Exhibitions zurück, die all diese enthalten. Zuerst werden alle Ids nach ihrer Existenz überprüft. Danach wird ein SQL-Befehl erstellt, der alle Exhibitions mit der ersten Kategorie der Id-Liste in eine weitere Liste speichert. Anschließend wird eine Schleife erstellt, die durch die Exhibitions-Liste durchläuft und überprüft, ob die restlichen gesuchten Kategorien Teil der aktuellen Exhibition in der Schleife sind. Falls ja, werden sie in eine neue Liste gesichert, die am Ende zurückgegeben wird.

Listing 22: Beispiel für neue Exhibition

```
public List<ExhibitionWithUserDTO> getByCategoryIds(List<Long> ids) {
```

```
2       String sql = "select new
            org.threeDPortfolioGallery.workloads.dto.ExhibitionWithUserDTO(e,
            u.user_name, u.icon_url) " +
3               "from Exhibition e join e.user u left join e.categories c where c.id
                    in :categoryid";
4       TypedQuery<ExhibitionWithUserDTO> q = getEntityManager()
5               .createQuery(sql
6                       , ExhibitionWithUserDTO.class
7               );
8       q.setParameter("categoryid", ids.get(0));
9       List<ExhibitionWithUserDTO> exhibitions = q.getResultList();
10      List<Long> catIds = new LinkedList<>();
11      var ret = new LinkedList<ExhibitionWithUserDTO>();
12      for (var exhibition: exhibitions ) {
13          if(exhibition.getExhibition().categories != null){
14              catIds.clear();
15              for (var cat: exhibition.getExhibition().categories) {
16                  catIds.add(cat.id);
17              }
18              if(new HashSet<>(catIds).containsAll(ids)){ // statt
                    catIds.containsAll(ids) wegen performance, ye
19                  ret.add(exhibition);
20              }
21          }
22      }
23      return ret;
24  }
```

3.6.1 Records und DTOs

Records wurden mit JDK 14 hinzugefügt. Sie sind Klassen, dessen Werte gesetzt, aber nicht mehr verändert werden können. Für die Definition solch einer Klasse wird nur das Stichwort *record* benötigt, zusammen mit Datentyp und Namen der gewünschten Felder. [36]

Für diese Arbeit wurde anfangs der Record in Codeblock 23 verwendet, um Daten zurückzuwerfen. Durch nicht ausreichender Java-Version in der Cloud-Umgebung, wurde die die Quarkus-Applikation im Nachinein für eine Kompabilität mit Java 11 verändert, was Records nicht mehr zuließ.

Listing 23: Verworfener Exhibition Record

```
1   public record ExhibitionWithUserRecord(Exhibition exhibition, String user_name,
        String user_icon_url) {
2   }
```

DTOs hingegen können mittels Set-Methoden ihre Werte wieder ändern. Dies ist nicht notwendig beim Auslesen der Datenbank, jedoch nützlich beim Anlegen neuer Objekte. Durch eine Verwendung des Lambok Frameworks werden die üblichen Get- und Set-Methoden automatisch angelegt. [37]

Listing 24: Exhibition DTO

```
1   @Data
2   public class AddExhibitionDTO {
3       String thumbnail_url;
4       String title;
5       String description;
```

```
6       Long room_id;
7       Long user_id;
8       Long[] category_ids;
9       AddExhibitDTO[] exhibits;
10  }
```

3.6.2 Verwaltung restlicher Entitäten

Für die Entitäten Category, Exhibit, Theme und Room wurden keine komplexen Methoden benötigt. Dies liegt unter anderem daran, dass sich das Team dazu entschlossen hat, Nutzenden keine Möglichkeit zu geben, neben Exhibitions und Exhibits, neue Objekte anzulegen. Deswegen bestehen die restlichen Resource-Klassen überwiegend aus GET-Methoden, wo entweder alle Entitäten aus der Datenbank gelesen werden, oder nur ein spezifisches Objekt mithilfe einer Id.

3.6.3 Dateiverwaltung

Für die Verwaltung der Hochgeladenen Dateien gibt es folgende Methoden:

- **GET** downloadFile(String exhibition, String fileName)
- **GET** downloadImageFile(String exhibition, String fileName)
- **POST** uploadFile(MultipartFormDataInput input)

Für den Download einer Datei wird dessen Pfad als Parameter mitgeschickt. Beim Aufruf der Methode *downloadFile* wird dieser Pfad überprüft. Falls eine Datei gefunden wird, wird mittels *Tika* eine Response erstellt, die das File zurückwirft.

Die *downloadImageFile*-Methode funktioniert gleich, nur spezialisiert sie sich auf Bild-Dateien. Sie hat ebenso *@Produces("image/png")* festgelegt.

Listing 25: downloadImageFile-Methode

```
1  File file = new File(FILE_PATH + exhibition + "/" + fileName);
2  if (!file.exists()) {
3      return Response.noContent().entity("file not found").build();
4  }
5  return Response.ok(file).header("Content-Disposition", "inline;filename=" +
       fileName).build();
```

Der File-Upload geschieht während der Bearbeitung der Exhibition im Frontend. Nach jeder Dateiauswahl des Users auf der Webseite, wird der Endpoint zuständig für den Fileupload, also die Methode *uploadFile*, aufgerufen. Dieser erwartet sich, anders als die anderen POST-Methoden, kein JSON-Objekt, sondern eine Datei als Parameter. Wiedermals wird dies fesgelegt mittels der *@Consumes*-Annotation, mit *"multipart/form-*

data" als Wert. Das Limit der akzeptierten Dateien wird in den *application.properties* festgelegt.

```
quarkus.http.limits.max-body-size = 300m
```

Diese Methode ist unpraktisch, falls der Nutzende im Nachhinein hochgeladene Objekte löschen möchte. Denn es wurde keine Methode implementiert, die überflüssige Dateien entfernt. Da diese Arbeit in einem kleinen Rahmen entwickelt wurde, ohne Absicht für eine Weiterführung nach dem Schulabschluss, wurde auf derartige Ressourcen bewusst keine Rücksicht genommen.

Für das eigentliche Abspeichern einer Datei, ist die Methode *writeFile()* zuständig (siehe Abb. 26). Diese wird aufgerufen, wenn der Endpoint aufgerufen wird und eine Datei mitgegeben wurde. Der erste Parameter der Methode, das Byte-Array, wird mittels eines Inputstreams erstellt. Dieser Prozess ist eine Art Boilerplate Code. Der Filename wird in einer alternativen Methode erstellt. Dabei werden Leerzeichen entfernt und der Name noch etwas abgeändert, damit nicht mehrere Files unter dem selbem Namen abgespeichert sind. Dafür wird der ursprüngliche Name der Datei auf Leerzeichen untersucht. Sobald diese entfernt wurden, wird der abgeänderte Name zurückgegeben.

Nachdem das File abgespeichert wurde, wird dessen Pfad mit dem neuen Namen in die Response mitgegeben. Diese Antwort wird dann als Wert erwartet als URL, beim Anlegen neuer Exhibits.

Listing 26: Hochladen der Dateien

```
private void writeFile(byte[] content, String filename) throws IOException {
    File file = new File(filename);
    if (!file.exists()) {
        file.createNewFile();
    }
    FileOutputStream fos = new FileOutputStream(file);
    fos.write(content);
    fos.flush();
    fos.close();
}
```

3.7 Tests

In diesem Kapitel werden die verschiedenen Testverfahren aufgezählt und etwas erläutert.

3.7.1 Während der Entwicklung

Tests sind eine gute Möglichkeit, zu gewährleisten, dass die Applikation fehlerfrei ihre Aufgaben erledigt. Diese sollten eine breite Auswahl von möglichen Situationen abdecken.

Für das Testen während der Entwicklung wurde die Extension *swagger-ui* verwendet. Erreichbar ist die Testresource unter *http://localhost:8080/q/swagger-ui/*. In dem lila Rechteck lässt sich der Pfad erkennen, während sich in dem roten Rechteck der Button zum Ausprobieren befindet. Nachdem das "Try it out"geklickt wurde, können die benötigten Werte für den Endpoint eingegeben werden (siehe Abb. 19). Der Pfad kann bei GET-Requests einfach in einem Browserfenster eingegeben werden und liefert dieselbe Antwort, wie das Tool.

Für die Entwicklung an JWT-geschützten Endpoints wurde der grün eingerahmte Button aus Abbildung 19 benötigt. Hier können Token eingegeben werden, wodurch normal getestet werden kann. Um diesen JWT zu erhalten, wurde ein separater Endpoint zum Testen entwickelt.

```
1  @GET
2  @Produces(MediaType.TEXT_PLAIN)
3  public Response getJwt(){
4      String jwt = jwtService.generateJwt();
5      return Response.ok(jwt).build();
6  }
```

In Abbildung 20 lässt sich erkennen, dass die generierte Abfrage für numerische Werte standardmäßig *0* und alphabetische Sequenzen *string* einsetzt. Zweiteres ist kein Problem, da jedoch beim Anlegen jedes neuen Objekts die Id generiert wird, ist es einfacher, die Id aus dem Request zu entfernen. Falls jedoch ein bestimmter Wert für dieses Attribut gewünscht ist, muss die Einzigartigkeit der angelegten Entitäten beachtet werden. Ansonsten werden neue Objekt womöglich nicht gespeichert.

3.7.2 Nach der Entwicklung

In dieser Arbeit wird mit JUnit 5 und Mockito gearbeitet. Mockito ermöglicht es, die Repositories zu rekonstruieren. Diese Rekonstruktion werden *Mocks* genannt. [38]

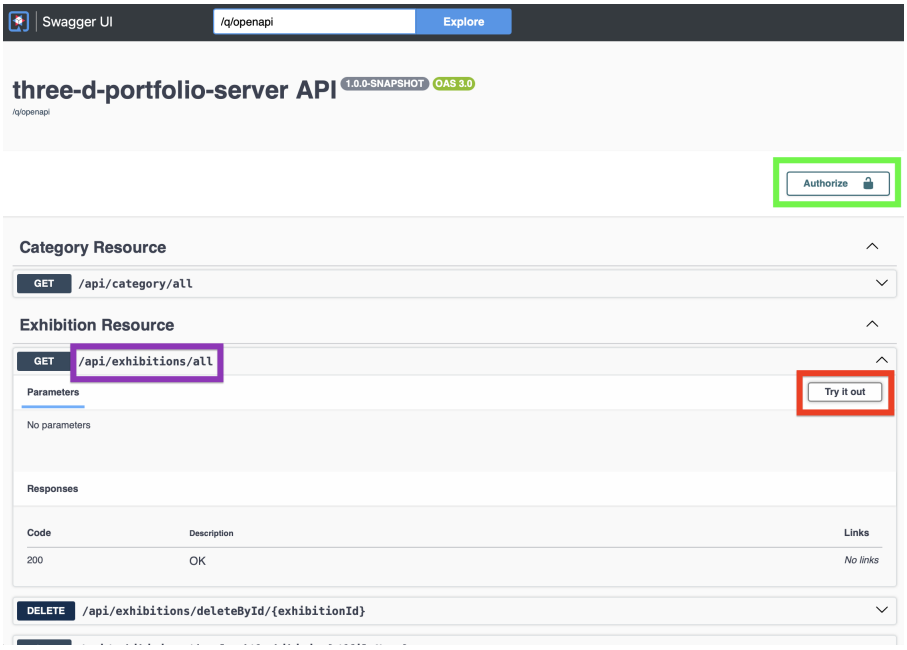


Abbildung 19: Übersicht der erstellten Schnittstellen

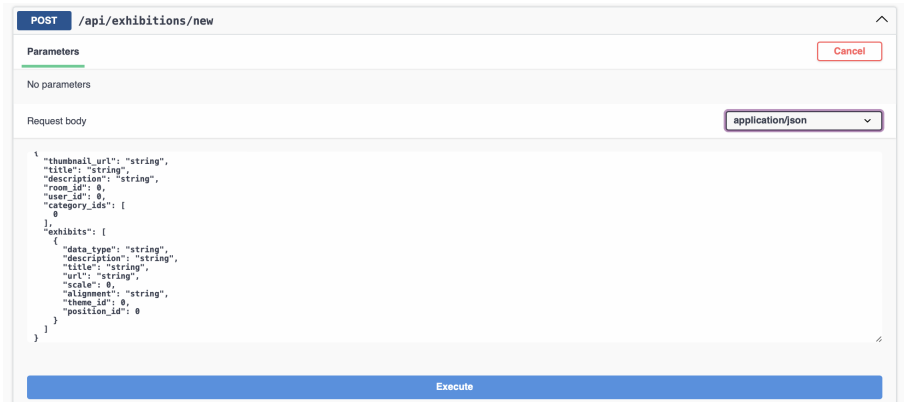


Abbildung 20: Automatisch generierte JSON-Anfrage

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5-mockito</artifactId>
    <scope>test</scope>
</dependency>
```

Tests in Quarkus werden mit *@QuarkusTest* gekennzeichnet. Mocks werden durch *@InjectMock* erstellt. Diese stellen alle Methoden aus den gemockten Repositories zur Verfügung. Zusätzlich besteht die Möglichkeit zu definieren, dass eine Methode vor jedem Test ausgeführt werden soll mittels *@BeforeEach*. Die einzelnen Tests werden zusätzlich mit *@Test* annotiert. [38]

Durch die Priorisierung des Testvorganges während der Entwicklung, wurden die zusätzlich angelegten Testklassen nur für die wichtigsten Endpoints genutzt. Dazu zählt beispielsweise die Entität Exhibition. Im unteren Beispiel wird ein Requst auf den Endpoint *api/exhibits/* erstellt, wobei keine Exhibition vorhanden ist. Dadurch wird der Statuscode 404 erwartet.

```
@Test
public void testGetExhibitByIncorrectId() {
    given()
            .when()
            .pathParam("exhibitionId", 1L)
            .get("/api/exhibitions/{exhibitionId}")
            .then()
            .statusCode(404);
}
```

3.8 Hosten auf einer Cloud

Bisher zu diesem Zeitpunkt wurde die Applikation immer lokal ausgeführt. Dies führte zu einer Optimierung auf den regelmäßig verwendeten Geräten, jedoch zu Problemen bei neuen, aufgrund der benötigten Installationen und Einrichtung der benutzten Technologien. Ein einfacher Startvorgang des Projektes benötigt *Java*, *Maven*, *PostgreSQL*, *Angular* und der *node package Manager*. Das erschwert die Nutzung der Applikation.

Ebenso benötigt jeder Interessierte einen Zugang auf die GitHub Repositories, welche privat gestellt sind. Um das Projekt nun für jeden zur Verwendung zu stellen, muss dieses online verfügbar gemacht werden.

3.8.1 Vorbereitung

Bevor die Applikation hochgeladen werden kann, müssen einige Änderungen am Code vorgenommen werden. Zum einen muss in den *application.properties* einiges für die Production festgelegt werden. Die Datenbankwerte müssen den Konfigurationen des PostgreSQL-Services entsprechen. Genauso muss der Pfad für die Dateiverwaltung definiert werden auf der Cloud. Zeile 7 legt den Uploadpfad der Dateien fest.

Listing 27: application.properties für Produktion

```
1 %prod.quarkus.datasource.jdbc.url=jdbc:postgresql://postgres:5432/demo
2 %prod.quarkus.datasource.username = demo
3 %prod.quarkus.datasource.password = demo
4 %dev.quarkus.hibernate-orm.database.generation=drop-and-create
5 %dev.quarkus.hibernate-orm.sql-load-script=import.sql
6 quarkus.container-image.name=3dserver
7 %prod.three-d.file.upload.path=/srv/upload/files/
```

In dieser Arbeit wurde für das Deployment die LeoCloud verwendet. Als Schüler oder Schülerin der HTBLA Leonding kann diese nach simpler Registrierung ohne Zahlungsvorgang genutzt werden. Die Cloud nutzt hauptsächlich Kubernetes, wodurch eine Installation der zugehörigen CLI *kubectl* fundamental ist.

Die LeoCloud verfügt über ihr eigenes Login Portal. Nach erfolgreicher Anmeldung wird eine Startseite angezeigt mit einem Bereich namens *Mein Konto*. Dieser bildet den Namen des Namespaces ab, den Download eines demonstrativen Services, sowie andere Informationen. Der wichtigste Teil dieses Fensters ist der Button *Mein Dashboard*.

Nach dessen Betätigung wird ein weiteres Fenster ersichtlich, was ein kurzes Tutorial anzeigt, wie auf das Dashboard zugegriffen werden kann. Das Kontrollzentrum lässt sich

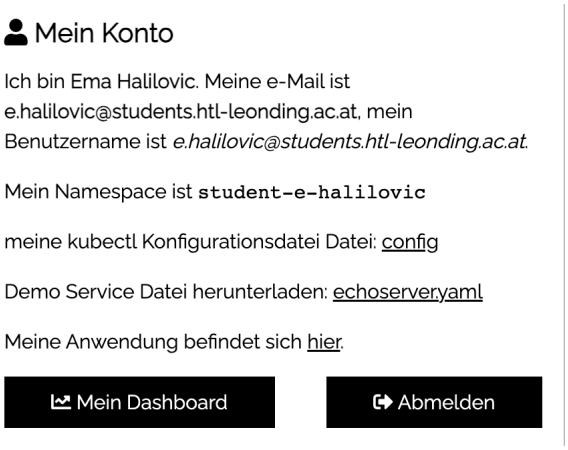


Abbildung 21: Persönlicher Bereich in der LeoCloud

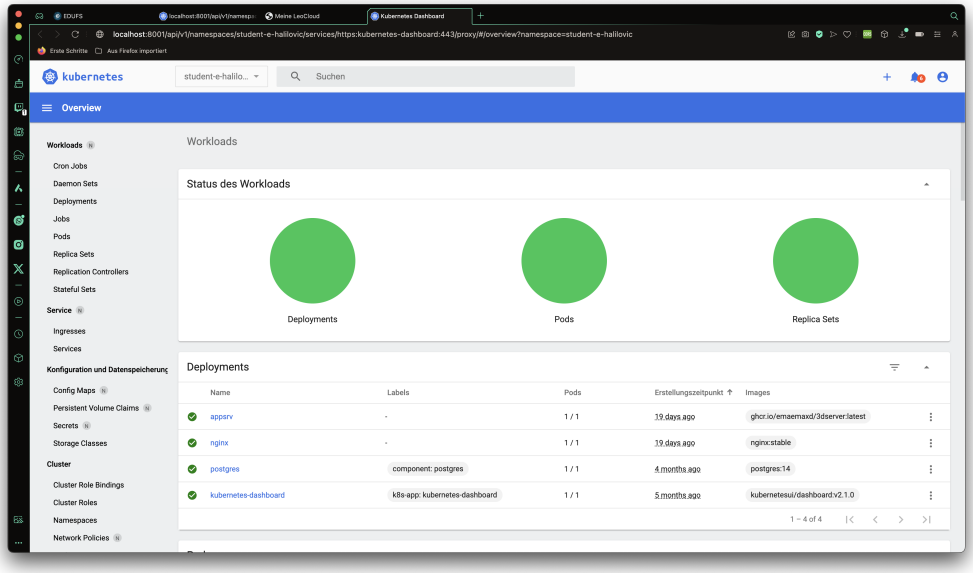


Abbildung 22: Dashboard der LeoCloud

nur anzeigen, wenn im Terminal der Befehl aus 28 ausgeführt wurde. Das Dashboard listet alle einzelnen Deployment auf, und ermöglicht es, Details anzuzeigen.

Listing 28: Ausführung des Proxys für Kubernetes

```
1  kubectl proxy
```

Neben der Vorbereitung des Dashboards, wird ebenso ein GitHub Token für die Nutzung des GHCR benötigt. Dieser wird erstellt in den Developer Settings der Accounteinstellungen. Nach einer erfolgreichen zwei-faktor-Authentifizierung, können nun alle Rechte des Tokens angegeben werden. Für diesen Fall werden nur die in Abbildung 23 rosa eingerahmten Berechtigungen benutzt.

Nach Erstellung wird im Terminal Befehl 29 eingegeben. Im Terminal werden dann die weiteren Schritte beschrieben (siehe Abb. 24). Bei der Passwortanforderung muss der

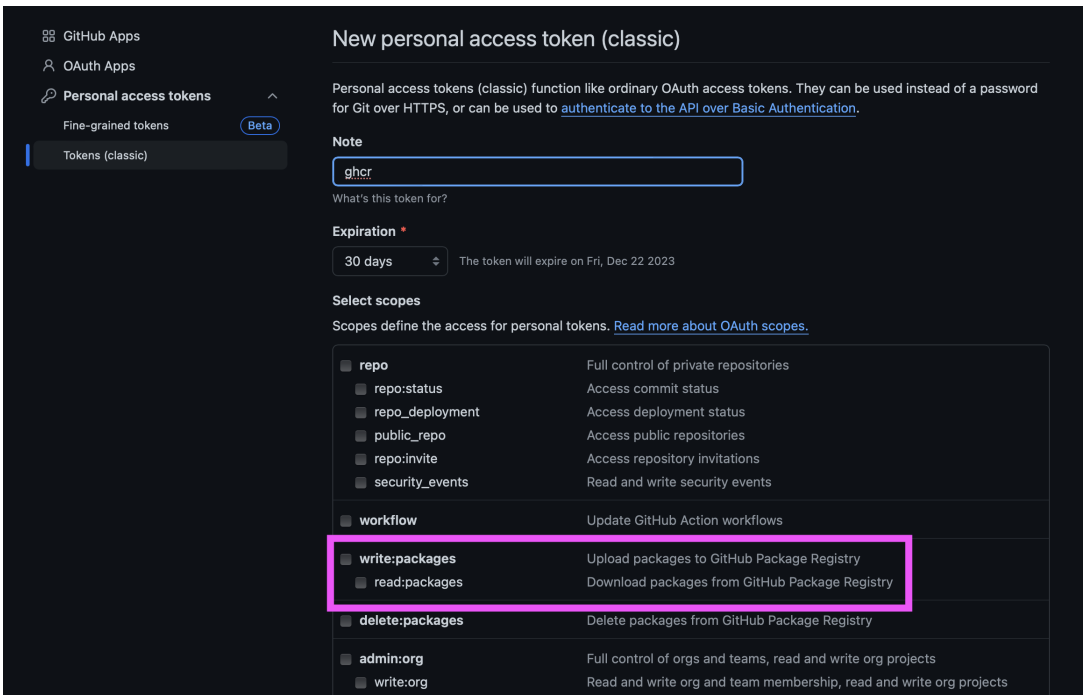


Abbildung 23: Erstellung eines neuen Tokens

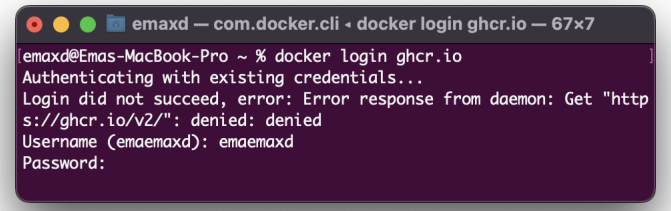


Abbildung 24: Login für das GHCR

in GitHub erstellte Token angegeben werden. Wichtig ist zu beachten, dass Docker am Gerät gestartet sein muss.

Listing 29: GHCR Login

```
1  docker login ghcr.io
```

3.8.2 Umsetzung

Dafür wurde ein Ordner aus dem Unterricht mit Herr Christian Aberger als Vorlage genommen namens *k8s*. In diesem waren allgemeine Dateien, wie *namespace.yaml*, schon angelegt, jedoch waren Anpassungen noch benötigt. Der Codeblock aus 30 zeigt den ganzen Inhalt dieses Files an. In dem vorher erwähnten File wird Zeile 4 in 30 mit dem angezeigten Namespace aus Abbildung 21 ersetzt.

Listing 30: Namespace konfiguration in namespace.yaml

```
1 apiVersion: v1
2 kind: Namespaces
3 metadata:
4   name: student-e-halilovic
```

Das Hauptelement des Deployments ist das *build.sh*-File. Es wird im Projekt-Root-Verzeichnis angelegt, da es durch die zentrale Position auf alle Ordner einfach zugreifen kann. In dieser Datei werden Prozesse für den Upload, sowie dafür benötigte Variablen, definiert. Wichtig sind Variablen, die den kompletten Namen des Packages definieren, sowie den GitHub-User. Dabei muss beachtet werden, dass Docker-Packagenamen keine Großbuchstaben enthalten dürfen. Um die Variablen für andere Skripten zur Verfügung zu stellen, werden diese exportiert. Die Umsetzung dieses Prozesses erfolgt durch die Befehle in Codeausschnitt 31.

Listing 31: Anfang der build.sh-Datei

```
1 BASE_HREF=${BASE_HREF:-"/e.halilovic/"}
2 GITHUB_USER=${GITHUB_USER:-emaemaxd}
3 BACKEND_IMAGE_NAME=ghcr.io/$LC_GH_USER_NAME/3dserver:latest
4 export GITHUB_USER
5 export BACKEND_IMAGE_NAME
```

Nachdem die Hauptkonfiguration definiert wurde, können nun die Hauptkomponenten Schritt für Schritt für den Upload vorbereitet werden. Zuerst wird mit dem Befehl *pushd* der aktuelle Pfad in den Verzeichnisstapel gelegt und in die Quarkus-Anwendung hineingewechselt. Anschließend wird die Applikation mit Maven in ein Package kompiliert und in einen neuen Ordner hineinkopiert. Mithilfe von Docker wird nach diesem Schritt ein Image erstellt unter dem Namen *3dserver*. Dieses wird von Docker benutzt, um ein weiteres Image zu erzeugen, nur dieses Mal unter dem Namen der vorher definierten Variable *BACKEND_IMAGE_NAME* (siehe Abb. 31 Zeile 6). Zuletzt wird das Image auf das GHCR gepusht und der Pfad des Shell-Skripts auf den Ausgangswert zurückgesetzt. Dies alles beschreibt die Befehle in Codeausschnitt 32.

Listing 32: Quarkus-Teil in der build.sh-Datei

```
1 pushd server/three-d-portfolio-server
2 mvn clean package
3 mkdir -p target/deploy
4 cp target/*-runner.jar target/deploy/
5 docker build --tag 3dserver --file ./src/main/docker/Dockerfile ./target/deploy
6 docker image tag 3dserver $BACKEND_IMAGE_NAME
7 docker push $BACKEND_IMAGE_NAME
8 popd
```

Danach wird die Angular-Applikation vorbereitet. Hier wird ebenso in das dafür zugehörige Verzeichnis gewechselt. Mittels dem node package Manager werden alle Abhängigkeiten installiert, bevor die Anwendung gestartet wird. Der Befehl in Codeabschnitt

33 Zeile 3 konfiguriert zusätzlich noch, dass mit Produktionskonfiguration gearbeitet werden soll. Ebenso wird definiert, unter welchem Pfad die Anwendung verfügbar sein wird.

Listing 33: Frontend-relevanter Teil der build.sh-Datei

```
1 pushd 3D-Portfolio-Gallery/Gallery
2 npm install
3 npm run build -- --configuration production --base-href $BASE_HREF
4 popd
```

Nachfolgend wird das *deploy.sh*-File im *k8s*-Ordner (siehe Codeabschnitt 34) ausgeführt. Diese definiert gleich in Zeile 1, dass nach einem Fehler das Skript abgebrochen werden soll. Zeile 3 sorgt dafür, dass die Variable durch ihren tatsächlichen Wert ersetzt wird. Die nachfolgenden Befehle legen Konfigurationen für den Kubernetes-Cluster fest.

Listing 34: k8s deploy.sh-Datei

```
1 set -e
2 mkdir -p target
3 envsubst '\$BACKEND_IMAGE_NAME' < appsrv.yaml > ./target/appsrv.yaml
4 kubectl delete -f appsrv.yaml || echo "appsrv not deployed yet"
5 # ...
6 kubectl apply -f busybox-job.yaml
7 kubectl apply -f ingress.yaml || echo "no ingress"
8 kubectl rollout restart deployment/nginx || echo "no nginx yet"
9 kubectl rollout restart deployment/appsrv || echo "no appsrv yet"
```

Die angewendeten Dateien enthalten alle näheren Konfigurationen für die einzelnen Pods. Der unten angeführte Codeblock stammt aus der *appsrv.yaml*-Datei, die sehr relevant ist für die Erstellung des Quarkus Applikationsserver. In ihr definiert man unter anderem, welches Image gezogen werden soll, unter welchem Pfad die Dateien gesichert werden, sowie unter welchem Port die Anwendung erreichbar ist. Das Image, welches in der Variable *BACKEND_IMAGE_NAME* steht, wird bei Ausführung der *build.sh*-Datei das Package privat auf GitHub hochgeladen. Damit der Applikationsserver darauf zugreifen kann, muss es jedoch manuell in den Packagesettings veröffentlicht werden (siehe Abb. 25).

Listing 35: Ausschnitt aus appsrv.yaml-Datei

```
1     apiVersion: apps/v1
2     kind: Deployment
3     metadata:
4       name: appsrv
5       namespace: student-e-halilovic
6     spec:
7       replicas: 1
8       selector:
9         matchLabels:
10          app: appsrv
11      template:
12        metadata:
13          labels:
14            app: appsrv
15        spec:
16          containers:
```

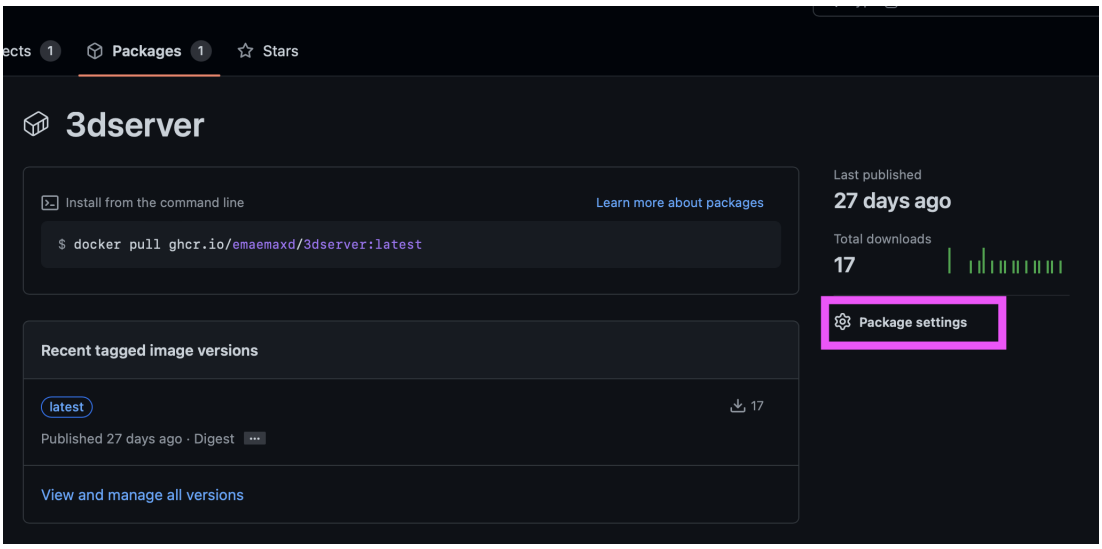


Abbildung 25: Ansicht des erstellten Packages in GitHub

```
17              - name: appsrv
18                image: $BACKEND_IMAGE_NAME
19                imagePullPolicy: Always
20                ports:
21                  - containerPort: 8080
22                volumeMounts:
23                  - name: upload
24                    mountPath: /srv/upload
25            volumes:
26              - name: upload
27                persistentVolumeClaim:
28                  claimName: appsrv-upload
29      ---
30      apiVersion: v1
31      kind: Service
32      metadata:
33        name: quarkus
34        namespace: student-e-halilovic
35      spec:
36        ports:
37          - port: 8080
38            targetPort: 8080
39            protocol: TCP
40        selector:
41          app: appsrv
```

Für Angular wird ein nginx-Server benötigt. Dieser hostet das Frontend und leitet die Requests an die Quarkus Anwendung weiter. Anfangs sieht das *nginx.yaml*-Skript dem *appsrv.yaml* ähnlich, jedoch wird am Ende noch ein Proxy eingerichtet.

Listing 36: Ausschnitt aus nginx.yaml-Datei

```
1  # ...
2  data:
3      default.conf: |
4      server {
5          listen 80;
6          rewrite_log on;
7          error_log /dev/stdout debug;
8          root /usr/share/nginx/html/gallery;
9          index index.html;
10         try_files $uri $uri/ /index.html =404;
11
12         location /api/ {
13             proxy_pass http://quarkus:8080;
14             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
15             proxy_set_header X-Real-IP $remote_addr;
16             proxy_set_header Host $host:$server_port;
```

```
17          }
18      }
```

Nach dem Skript wird eine Funktion aufgerufen, die auf den Start des Pods wartet. Dies geschieht durch die Ausführung eine Schleife, die in jedem Durchgang überprüft, ob ein Wert in der Variable *KNIFE_POD* steht. Falls nicht, kommt es zu einem neuen Durchgang, wobei die Variable neu geschrieben, sodass die Schleife nicht endlos läuft. Ebenso wird bei jeder Runde für eine Sekunde gewartet. Dies geschieht durch den Befehl in Zeile 3 in Codeausschnitt 37. Hier werden die Pods wiedergegeben und zurechtgeschnitten, sodass sie am Ende in der Variable exportiert werden.

Listing 37: Knife-Funktion in der build.sh-Datei

```
1  KNIFE_POD=""
2  findPod() {
3      KNIFE_POD=$(kubectl -n $NAMESPACE get pods|grep -i Running|grep knife|cut -d\
           -f 1)
4  }
5  waitForPod() {
6      local pod=""
7      while [ "$KNIFE_POD." == "." ]; do
8          findPod $1
9          kubectl -n $NAMESPACE get pods | grep knife
10             echo "wait for knife"
11             sleep 1
12     done;
13 }
14 waitForPod knife
15 export KNIFE_POD
16 export NAMESPACE
```

Nach dieser Funktion wird im Angular-Verzeichnis die *deploy.sh*-Datei ausgeführt. Diese Kopiert das dist-Verzeichnis auf den Namespace Pod.

Listing 38: deploy.sh-Datei für Angular

```
1  set -e
2  kubectl -n $NAMESPACE  exec $KNIFE_POD -- rm -rf /srv/demo /srv/dist
3  pushd ./dist
4      kubectl -n $NAMESPACE  cp * $KNIFE_POD:/srv/
5  popd
```

Nach Angular werden die Quarkus Dateien auf den Namespace geladen.

Listing 39: Kopieren der Datei für Quarkus

```
1  pushd server/three-d-portfolio-server/target
2  kubectl -n $NAMESPACE cp files/ $KNIFE_POD:/mnt/
3  popd
```

Der letzte Befehl löscht die Resource *busybox-job.yaml* aus dem Kubernetes-Cluster, die in Codeabschnitt 34 definiert wurde.

Listing 40: letzter Befehl der build.sh-Datei

```
1  kubectl delete -f busybox-job.yaml
```

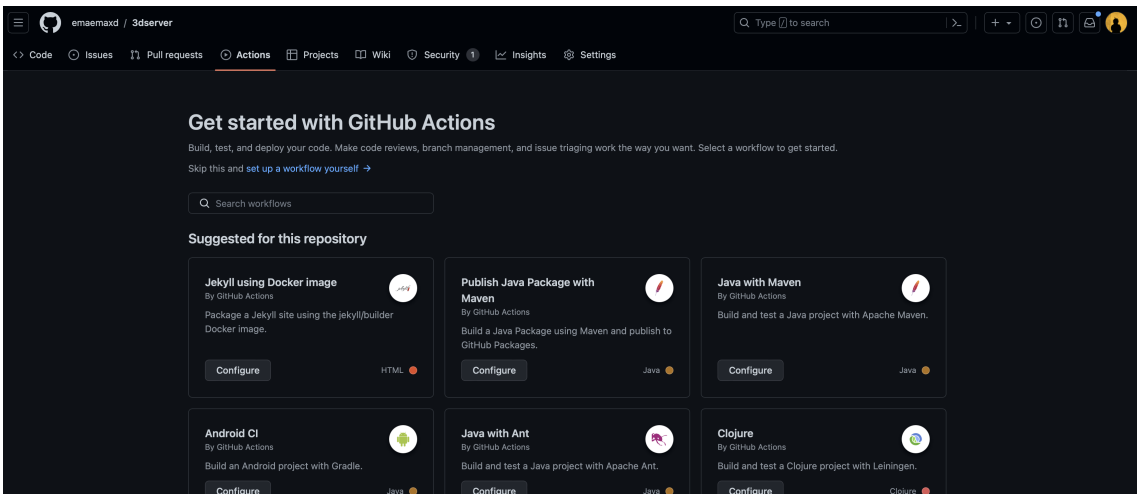


Abbildung 26: GitHub Actions Benutzeroberfläche

3.9 Continious Integration und Deployment

Zusätzlich war eine Implementation einer Continious Integration, sowie eines Continious Deployments geplant. Die beiden Begriffe lassen sich mit CI/CD abkürzen. Sie sind für eine Automatisierung zuständig während der Softwareentwicklung.

3.9.1 Continious Integration

Eine Continious Integration bedeutet, dass die Applikationsänderungen mittels mehrerer Überprüfungen auf ihre Funktionalität getestet wird. Dies ist besonders praktikabel, wenn im Projekt mehrere Entwickler Änderungen im Code vornehmen. Eine CI-Pipeline stellt sicher, dass diese keinen Konflikt verursachen bei der Zusammenführung dieser verschiedener Versionen entsteht. Dies geschieht, durch Ausführung des Programms und eine Validierung der Tests. [39]

Solch eine Automatisierung kann durch mehrere Arten verwirklicht werden, wie beispielsweise durch GitHub Actions. In jedem GitHub Repository gibt es eine Registerkarte namens *Actions*, diese bietet Vorschläge zur Erstellung solch einer Pipeline (siehe Abb. 26).

Jede Pipeline, oder auch Prozesskette, sind Dateien, welche in einem Ordner namens *workflows* angelegt werden. In diesen Dateien sind Prozesse und Konfigurationen definiert, wie beispielsweise der Name der Pipeline. Ebenso wird definiert, wann diese ausgeführt werden. Im Codeausschnitt 41 in Zeile 3 bis 7 wurde definiert, dass die nachfolgenden Steps bei jedem Push- und Pull-Request am Main Branch ausgeführt werden. Ebenso wird der benötigte Runner definiert in Zeile 11. Danach werden die

einzelnen Schritte definiert, um die CI für die Applikation zu erstellen, zum Beispiel die Installation aller node Module. In diesem Fall sind alle Steps Teil von einem einzigen Job namens "build".

Listing 41: Pipeline einer CI

```
name: CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  # ...
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Install all node modules
        run: npm i
        working-directory: ./3D-Portfolio-Gallery/
    # ...
```

3.9.2 Continious Delivery

Eine CD kann als ein Upgrade der CI gesehen werden. Das Ziel dieser ist es, die Applikation für die Produktion bereitzustellen. Dies geschieht ebenfalls durch eine Pipeline, welche die Anwendung Schritt für Schritt durchtestet und anschließend auf den Produktionsserver lädt. [39]

In diesem Projekt wäre die Pipeline für die Ausführung der *build.sh*-Datei zuständig gewesen. Zusätzlich hätte die Entfernung der Pods definiert werden müssen.

Während der Entwicklung kam es zu dem Fazit, dass eine Konfiguration einer CI/CD Pipeline als nicht nötig empfunden wurde. Dies ist einerseits, da nur eine Person an dem Backend arbeitete und somit keine zweite Validierung der Software nötig war, da diese schon manuell ausgeführt wurden. Andererseits hätte es eine große Komplexität bedeutet, da Frontend und Backend auf unterschiedlichen Repositories entwickelt wurden und erst zum Schluss vereint.

4 Zusammenfassung

In diesem Kapitel wird ein Überblick über das Projekt gegeben, wobei ebenso auf die verschiedenen Herausforderungen, die im Laufe des Projekts aufkamen, eingegangen wird.

4.1 Herausforderungen in Quarkus

Schon zu Beginn der Entwicklung des Quarkus-Servers kamen Hindernisse zustande. Das Gerät zur Entwicklung warf bei jedem Kompilierungsversuch Fehlermeldungen, die Schritt für Schritt gelöst werden mussten. Das Problem ließ sich nach einigen Änderungen der installierten Entwicklertools, sowie neuen Installationen von Java und Maven, lösen. Jedoch nahm es viel Zeit in Anspruch, und führte dazu, dass serverseitige Anforderungen nicht zeitgemäß erfüllt wurden.

Die Methode *getExhibitionByCategories* hat sich aufgrund der Many-to-Many-Beziehung als große Herausforderung dargestellt. Quarkus-Extensions bieten keine Möglichkeit die gewünschte Aufgabe ressourcensparsam umzusetzen. Da für die Applikation keine zukünftigen Pläne bestehen, wurde das Problem mittels Java-Listen gelöst.

Neben der Entwicklung wurde der Code mittels Javadoc dokumentiert. Im Nachhinein lässt sich sagen, dass dies sich als nicht sonderlich nützlich herausgestellt hat, da es zeitaufwendig war. Von dem Zeitfaktor abgesehen, gab es keine weitere Person, die von dem erstellten Dokument profitiert hätte. Dadurch lässt sich behaupten, dass normale Kommentare ausgereicht hätten.

4.2 Herausforderungen im Deployment

Das Deployment der Anwendung stellte sich als eine sehr große Herausforderung dar. Neue Änderungen und Statusupdates der LeoCloud wurden nicht online geteilt, was die Nutzung als ehemaliger Schüler oder ehemalige Schülerin deutlich erschwerte.

Ebenso stellte sich das Deployment verschiedener Services als unübersichtlicher fest als gedacht. Dadurch kam es zu mehreren Caching-Problemen, die nur durch sorgfältige manuelle Löschverfahren behoben werden konnten.

Beide Probleme konnten durch einen regelmäßigen Kontakt mit Herr Aberger Christian, dem Hauptentwickler der LeoCloud, minimiert werden.

4.3 Zielerreichung

Die zwei Hauptaufgaben wurden erreicht, wobei Meileinsteien nicht rechtzeitig abgeschlossen wurden.

Das Endergebnis dieser Arbeit bietet dem Frontend eine Datenbank zur Datenverwaltung zusammen mit ersten angelegten Objekten. Zusätzlich ist die Webseite auf der LeoCloud unter dem Link https://student.cloud.htl-leonding.ac.at/e.halilovic/home erreichbar.

Glossar

API Der Begriff steht für *Application Programming Interface*. Durch APIs können verschieden Applikationen miteinander kommunizieren, ohne die jeweils andere Implementierung wissen zum müssen. [40]

Boilerplate Code Boilerplate Code sind meist Codeblöcke, die an verschiedene Situationen anwenden kann. Meist werden diese nicht geändert, manchmal jedoch müssen Anpassungen gemacht werden. [41]

CD Continious Deployment

CI Continious Integration

CLI Command Line Interface

CRUD Create Replace Update Delete

DTO Data Transfer Object

ERD Der Begriff steht für *Entity Relationship Diagram*. Diese Diagramme visualisieren, welche Beziehungen verschiedenen *Entitäten* zueinander haben. [42]

Framework Frameworks bieten Funktionen an, ohne dass diese selbst implementiert werden müssen. Sie erleichtern das Programmieren innorm. [43]

GHCR GitHub Container Registry

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JDBC Java Database Connectivity

JPQL Jakarta Persistance Query Language

JSON JavaScript Object Notation

JVM Java Virtual Machine

JWT JSON Web Token

ORM Object Relational Mapper

REST Representational State Transfer

SQL Structured Query Language

SSOT Der Begriff steht für *Single Source of Truth*. Er beschreibt ein Konzept, dass jedes Mitglied dieselben Daten besitzt, sodass Lösungsideen immer aktuell sind. [44]

TTL Der Begriff steht für *Time To Live*. Der Wert dessen legt fest, wie lange bestimmte Daten am Server gespeichert werden. [45]

UML Unified Modeling Language

URI Uniform Resource Identifier

Literaturverzeichnis

[1] Quarkus, „SUPERSONIC/SUBATOMIC/JAVA," WebPage, 2023, letzter Zugriff am 03.09.2023. Online verfügbar: https://quarkus.io

[2] ——, „What is Quarkus?" WebPage, 2023, letzter Zugriff am 01.09.2023. Online verfügbar: https://quarkus.io/about/

[3] ——, „What is Quarkus?" WebPage, 2023, letzter Zugriff am 01.09.2023. Online verfügbar: https://quarkus.io/get-started/

[4] T. A. S. Foundation, „What is Maven?" WebPage, 2023, letzter Zugriff am 02.09.2023. Online verfügbar: https://maven.apache.org/what-is-maven.html

[5] C. Team, „What Is Maven?" WebPage, 2023, letzter Zugriff am 23.11.2023. Online verfügbar: https://www.codecademy.com/resources/blog/what-is-maven/

[6] Quarkus, „CONFIGURE DATA SOURCES IN QUARKUS," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://quarkus.io/guides/datasource

[7] G. Chehab, „What Is an ORM? How Does It Work? How Should We Use One?" WebPage, 2023, letzter Zugriff am 03.09.2023. Online verfügbar: https://www.baeldung.com/cs/object-relational-mapping

[8] Quarkus, „SIMPLIFIED HIBERNATE ORM WITH PANACHE," WebPage, 2023, letzter Zugriff am 07.09.2023. Online verfügbar: https://quarkus.io/guides/hibernate-orm-panache

[9] YourKit, „RESTEasy," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://resteasy.dev

[10] Quarkus, „RESTEasy Classic," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://quarkus.io/guides/resteasy

[11] M. Contributers, „HTTP request methods," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[12] S. Software, „REST API Documentation Tool | Swagger UI," WebPage, 2023, letzter Zugriff am 07.09.2023. Online verfügbar: https://swagger.io/tools/swagger-ui/

[13] J.-D. Kranz, „Was ist JUnit? Was sind JUnit Tests?" WebPage, 2020, letzter Zugriff am 14.11.2023. Online verfügbar: https://it-talents.de/it-wissen/junit/

[14] T. P. G. D. Group, „PostgreSQL: about," WebPage, 2023, letzter Zugriff am 02.09.2023. Online verfügbar: https://www.postgresql.org/about/

[15] T. P. G. D. Grou, „pgAdmin," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://www.pgadmin.org

[16] JetBrains, „Features overview," WebPage, 2023, letzter Zugriff am 02.09.2023. Online verfügbar: https://www.jetbrains.com/idea/features/

[17] ——, „Subscription Options and Pricing," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://www.jetbrains.com/idea/buy/?section=personal&billing=monthly

[18] Microsoft, „Visual Studio Code - Code Editing. Redefined," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://code.visualstudio.com

[19] ——, „Getting Started," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://code.visualstudio.com/docs

[20] A. Christian, „LeoCloud User Manual," WebPage, 2021, letzter Zugriff am 03.09.2023. Online verfügbar: https://cloud.htl-leonding.ac.at/user-manual.html?#_allgemeines

[21] Kubernetes, „Kubernetes," WebPage, 2023, letzter Zugriff am 23.11.2023. Online verfügbar: https://kubernetes.io

[22] Microsoft, „Visual Studio Code Kubernetes Tools," WebPage, 2023, letzter Zugriff am 24.11.2023. Online verfügbar: https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools

[23] H. Staff, „What Is GitHub, and What Is It Used For?" WebPage, 2016, letzter Zugriff am 24.11.2023. Online verfügbar: https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/

[24] N. Labs, „Über Notion," WebPage, 2023, letzter Zugriff am 03.09.2023. Online verfügbar: https://www.notion.so/de-de/about

[25] O. Object Management Group, „OMG® Unified Modeling Language® (OMG UML®)," WebPage, 2017, letzter Zugriff am 03.09.2023. Online verfügbar: https://www.omg.org/spec/UML/2.5.1/PDF

[26] PlantUML.com, „PlantUML – Ein kurzer Überblick," WebPage, letzter Zugriff am 03.09.2023. Online verfügbar: https://plantuml.com/de/

[27] jebbs, „PlantUML," WebPage, letzter Zugriff am 24.11.2023. Online verfügbar: https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml

[28] Auth0, „What is JSON Web Token?" WebPage, 2023, letzter Zugriff am 09.09.2023. Online verfügbar: https://jwt.io/introduction

[29] M. Howell, „Install Homebrew," WebPage, 2016, letzter Zugriff am 25.11.2023. Online verfügbar: https://brew.sh

[30] J. s.r.o., „Quarkus - IntelliJ IDEs Plugin | Marketplace," WebPage, 2023, letzter Zugriff am 26.10.2023. Online verfügbar: https://plugins.jetbrains.com/plugin/20306-quarkus

[31] baeldung, „Overview of JPA/Hibernate Cascade Types," WebPage, 2023, letzter Zugriff am 25.11.2023. Online verfügbar: https://www.baeldung.com/jpa-cascade-types

[32] Quarkus, „Using Transactions in Quarkus," WebPage, 2023, letzter Zugriff am 25.11.2023. Online verfügbar: https://quarkus.io/guides/transaction

[33] R. Grimm, „Patterns in der Softwarearchitektur: Das Schichtenmuster," WebPage, 2023, letzter Zugriff am 25.11.2023. Online verfügbar: https://www.heise.de/blog/Patterns-in-der-Softwarearchitektur-Das-Schichtenmuster-7941983.html

[34] Quarkus, „INTRODUCTION TO CONTEXTS AND DEPENDENCY INJECTION (CDI)," WebPage, 2023, letzter Zugriff am 25.11.2023. Online verfügbar: https://quarkus.io/guides/cdi

[35] ——, „Using JWT RBAC," WebPage, 2023, letzter Zugriff am 25.11.2023. Online verfügbar: https://quarkus.io/guides/security-jwt

[36] J. Albano, „Java 14 Record Keyword," WebPage, 2023, letzter Zugriff am 25.11.2023. Online verfügbar: https://www.baeldung.com/java-record-keyword

[37] T. P. L. Authors, „Project Lombok," WebPage, 2023, letzter Zugriff am 09.09.2023. Online verfügbar: https://projectlombok.org/#

[38] G. Andrianakis, „Mocking CDI beans in Quarkus," WebPage, 2020, letzter Zugriff am 26.11.2023. Online verfügbar: https://quarkus.io/blog/mocking/

[39] R. Hat, „Was ist CI/CD? Konzepte und CI/CD Tools im Überblick," WebPage, 2023, letzter Zugriff am 19.11.2023. Online verfügbar: https://www.redhat.com/de/topics/devops/what-is-ci-cd

[40] ——, „What is an API?" WebPage, 2022, letzter Zugriff am 21.08.2023. Online verfügbar: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces

[41] AWS, „Was ist Boilerplate-Code? - Boilerplate-Code erklärt – AWS," WebPage, 2023, letzter Zugriff am 09.11.2023. Online verfügbar: https://aws.amazon.com/de/what-is/boilerplate-code/

[42] V. Paradigm, „What is Entity Relationship Diagram (ERD)?" WebPage, 2023, letzter Zugriff am 24.08.2023. Online verfügbar: https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/;WWWSESSIONID=617AD2BCF587341A07E1E81EA0E2099C.www1

[43] C. Team, „What is a Framework?" WebPage, 2021, letzter Zugriff am 07.09.2023. Online verfügbar: https://www.codecademy.com/resources/blog/what-is-a-framework/

[44] Talend, „Single Source of Truth - What it is and Why You Want it Yesterday | Talend," WebPage, 2023, letzter Zugriff am 23.11.2023. Online verfügbar: https://www.talend.com/resources/single-source-truth/

[45] Kinsta, „What Is TTL (And How Do You Choose the Right One)?" WebPage, 2022, letzter Zugriff am 23.11.2023. Online verfügbar: https://kinsta.com/knowledgebase/what-is-ttl/

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Anhang